Universidad Carlos III de Madrid
Escuela Politécnica Superior

Ingeniería Técnica en Telecomunicación
Especialización en Telemática

PROYECTO DE FIN DE CARRERA

UNIAFFINITY

Autor: *Luis Cappa Banda*
Tutora: *Florina Almenares Mendoza*

Agradecimientos

A mis padres todo el trabajo y sacrificio que han destinado en mi educación. Quisiera guardar un silencio escrito, cómplice, que oculte tantos momentos de dificultad que me han enseñado a combatir siempre con tenacidad y fortaleza, enseñándome que no hay bien propio que valga más que el bien de un ser querido. Sois los mejores padres del mundo.

A mis abuelos, por todo el cariño y el amor que han depositado en mí. Guardo en mi memoria, intactos, todos aquellos recuerdos que de niño forjé con vosotros con golpes de humor, sonrisas y miradas cargadas de amor. Pienso en vosotros cada día.

A Patricia, mi mujer, por enseñarme la luz cuando todo se oscurecía y el camino se desmoronaba a mi paso. Has sido un punto de inflexión en mi vida seguido de otros tantos sin ningún aparte. Tenemos mucho que vivir y disfrutar, muchos sueños aún por cumplir, y espero vivir a tu lado hasta el último de mis días.

A mi perro, por tanto cariño desinteresado, por su bondad y nobleza, por tantas tardes y noches de estudio en las que siempre me ha acompañado fielmente sin esperar nada más que una caricia como gesto de recompensa.

A todos mis amigos y compañeros y, de nuevo, a mi mujer, pues sin haber compartido vuestro tiempo con el mío estudiar en la biblioteca habría sido insoportable, no habría tomado un chupito bañado en fuego en Ljubljana, no habría apostado con éxito cincuenta dólares al rojo en Las Vegas, no habría paseado bajo la sombra de los árboles de Central Park y, en definitiva, no habría descubierto esas partes de mí que sólo otra persona puede revelarte.

A mi enemigo, pues alguno tendré. Si lees estas líneas tan sólo déjame decirte algo: ¿no tenías nada mejor que hacer?

Resumen

El presente documento describe el proyecto de investigación *UniAffinity API*, cuyo objetivo reside en ofrecer una pasarela de comunicación HTTP hacia los sistemas de autenticación OAuth de Twitter y Facebook, dos de las redes sociales más relevantes del momento.

UniAffinity cuenta con un módulo de Minería de Datos de que permite obtener y almacenar datos personales de usuarios de las redes sociales. Además, *UniAffinity* implementa *UniAffinity API*, un interfaz de consumo externo que permite consultar los datos personales almacenados y establecer relaciones de parentesco entre usuarios para hallar usuarios similares.

Empleando modernas técnicas de modelado de datos NoSQL con tecnologías como Apache Solr y MongoDB, *UniAffinity API* implementa una arquitectura totalmente modular, distribuida, escalable que admite configuraciones en alta disponibilidad tolerante a fallos, haciendo frente con eficacia a los principales retos contemporáneos relacionados con el almacenamiento y consumo masivo de datos.

Abstract

The following document describes *UniAffinity API Project*, which objective resides on offering a universal HTTP gateway to Twitter and Facebook OAuth authentication systems, two of the most popular networks nowadays.

UniAffinity implements its own Data Mining module that allows it to obtain personal users data from Facebook and Twitter allowing to consume and store user information fetched from social networks. Furthermore, *UniAffinity* implements *UniAffinity API*, an interface that allows external data connections to retrieve user's information and to establish user relationships based on similar social network data tracks and patterns.

Using modern NoSQL data modeling technologies such as Apache Solr and MongoDB, *UniAffinity* API is built with a completely modular and distributed architecture, fail-tolerant and scalable that permits it to affront modern massive data storage and data consuming often called as '*Big Data*' challenges.

Contenido

Índice de figuras

Índice de tablas

Índice de ejemplos de código

1. Introducción

1.1 Motivación del proyecto

Día a día se generan ingentes cantidades de datos en la red de difícil análisis y manejo que dificultan su uso posterior. Las Redes Sociales han perdido su cariz innovador para ser una realidad que ocupa una parcela diaria de nuestra rutina diaria, con especial énfasis en un segmento de población joven-adulto.

Las redes sociales producen cada segundo cientos de gigas de información nacidas de las imágenes tomadas, comentarios vertidos, enlaces compartidos y otras muchas formas de opinión, debate o, en esencia, material informativo que toman diferentes canales pero que parten de un mismo origen: de uno mismo.

Uniaffinity API (Application Programming Interface) nace con el objetivo de analizar la información personal vertida en dos de las mayores redes sociales actuales, Twitter y Facebook, para que, por medio técnicas de Minería y Análisis de Datos a través de los diferentes REST (*Representational State Transfer*) APIs de ambas redes sociales, puedan almacenarse esta información extraída con un modelo de datos que permita relacionar a posteriori usuarios con otros usuarios, estableciendo lazos invisibles que la propia plataforma ayuda a mostrar.

Así, *Uniaffinity* implementa una REST API que permite a aplicaciones cliente consumir de estos recursos sociales y personales ya procesados, analizados y almacenados, sirviendo así de un complemento universal a cualquier producto de software que busque, por un lado, interactuar con redes sociales como Twitter y Facebook; por otro lado, poder conocer con más detalle el perfil, gustos, aficiones… de los usuarios que hagan uso de dicho software y fomentar que establezcan lazos de comunicación entre ellos antes inabordables.

UniAffinity servirá de agente intermedio entre las diferentes aplicaciones de *software* y las redes sociales, sirviendo de pasarela para este proceso de autenticación con Twitter y Facebook. Este proceso de autenticación se define como el diálogo entre una aplicación cliente y una aplicación servidora donde el cliente valida sus credenciales para que el servidor acceda a consumir los recursos o ejecutar acciones que estuviera reservadas sólo para usuarios identificados.

En los procesos de autenticación de usuarios a través de *UniAffinity*, además de interactuar con las redes sociales a modo de pasarela identificando al usuario en ellas, el sistema extraerá información de él que posteriormente será almacenada y podrá ser explotada y consumida por las aplicaciones de software a través de *UniAffinity API*.

Uniaffinity API es, en síntesis, una solución cómoda y práctica para el desarrollador que le permite contar con un módulo de autenticación de usuarios para el desarrollo de cualquier proyecto que lo requiera que, además, ofrece como grandes valores añadidos diferenciadores:

- La oportunidad de conocer más datos de sus usuarios a partir de la extracción de información de ellos de Twitter y Facebook.

- Fomentar relaciones entre usuarios similares en el ecosistema de un proyecto de software, aportando un componente social extra.

1.2 Objetivos

UniAffinity nace con el propósito de convertirse en una pasarela de comunicación hacia las redes sociales Twitter y Facebook que permitan a una aplicación cliente autenticar usuarios de su ecosistema con dichas redes sociales, extraer información personal de cada uno de ellos a través, procesarla y almacenarla persistentemente para luego ser consumida a través de *UniAffinity API*.

Para la consecución de esa meta se abordarán los siguientes objetivos:

- Investigar el desarrollo de aplicaciones para los ecosistemas de Twitter y Facebook.

- Investigar la especificación del protocolo de autenticación OAuth que emplean Twitter y Facebook en sus procesos de autenticación.

- Estudiar arquitecturas adoptadas por grandes empresas tecnológicas que fomenten la escalabilidad horizontal de un desarrollo de software que almacene grandes volúmenes de datos expuestos a ser procesados y consumidos posteriormente por un número creciente de personas al mismo tiempo.

- Estudiar el Estado del Arte de las diferentes alternativas de almacenamiento persistente, incluyendo bases de datos relacionales y bases de datos no relacionales.

- Investigar acerca de las diferentes aproximaciones que pueden adoptarse para desarrollar sistemas de recomendación y sistemas de búsqueda.

- Ofrecer una HTTP REST API accesible a través de autenticación que permita obtener información relativa a los usuarios que han sido registrados en el sistema y cuyos datos personales provenientes de Twitter y Facebook hayan sido procesados y almacenados.

1.3 Contenido de la memoria

El contenido de la memoria se estructura mediante los siguientes puntos de desarrollo:

Capítulo 2: Estado del arte

En este apartado se analiza y describen las diferentes tecnologías empleadas en el desarrollo del proyecto empleando un lenguaje y enfoque técnicos, incluyendo ocasionalmente puntualizaciones de carácter histórico que permitan al lector optar a un mejor y más profundo entendimiento de la temática tratada.

Capítulo 3: Descripción del sistema

Descripción detallada del sistema en conjunto analizando la disposición de los diferentes módulos que lo componen, sus funciones de qué modo interactúan entre sí. En este capítulo se detallarán los diferentes módulos que componen el sistema, así como también algunas decisiones de diseño relacionadas con funcionalidades deseadas y funcionalidades no deseadas (limitaciones).

Capítulo 4: Desarrollo del proyecto

En este capítulo ahondaremos aún más en los módulos que componen la arquitectura de *UniAffinity*, señalando algunos puntos importantes a tener en cuenta tales como el modelado de los datos o un detalle más profundo que explica de qué se compone el propio sistema de recomendación y qué partes involucradas hay en él.

Capítulo 5: Historia del proyecto

Línea temporal histórica y detallada de los diferentes hitos que componen el proyecto acompañadas de algunas referencias a impedimentos técnicos, anécdotas y otros contratiempos en el curso del trabajo. En este capítulo también se incluye un presupuesto acorde con las diferentes fases involucradas en el proyecto.

Capítulo 6: Conclusiones y líneas futuras de trabajo

Síntesis final del proyecto, análisis del producto final y sugerencias de líneas futuras de desarrollo o mejoras de lo ya existente.

Apéndices

Secciones de especial interés práctico como un *Manual de Instalación* del software de *UniAffinity* o un *Manual de desarrollo de Apps* tanto para Facebook como para Twitter.

2. Estado del Arte

2.1 Redes sociales

Internet ha evolucionado vertiginosa y rápidamente en los últimos años. En sus comienzos el desarrollo fue lento y estanco donde la información era compartida a una audiencia poco numerosa – aunque prometedora - a nivel mundial. Establecidas las bases del negocio por las operadoras de red, su evolución creció hasta alcanzar la categoría revolución, convirtiéndola en un medio de información global que hoy ocupa nuestro día a día como hábito irrenunciable.

Los primeros usuarios – y sus datos – carecían de movilidad: eran estáticos, poco flexibles, orientados a enviar, recibir, leer y escribir en un único lugar, convirtiendo el acceso a Internet en un uso atópico y de duración breve. La aparición de las redes de datos móviles y de los primeros teléfonos inteligentes ha permitido cambiar el modo de uso, y con él las bases del negocio, permitiendo a los usuarios hacer uso permanente de la Red.

Las redes sociales nacieron como una respuesta frente a la necesidad de mantener contacto permanente a toda la información de nuestro entorno o de nuestro interés, permitiéndonos estar permanente y perfectamente informados de las novedades a nuestro alrededor de manera inmediata.

En sus inicios, redes sociales como Facebook nacieron como una prueba de concepto para permitir compartir información con compañeros de la Facultad. Asimismo, Twitter nació de la idea de compartir información de manera breve y explícita, acercando el perfil del producto a un uso mucho más inmediato, cercano y reciente. La madurez de éstas y otras redes, junto con el crecimiento del mercado móvil, han hecho que las redes sociales sean hoy la primera fuente de información del mundo, tanto de corte periodístico como de índole personal.

En los dos puntos se tratarán con detalle las redes sociales Facebook y Twitter, incluyendo algunas referencias históricas sobre sus orígenes, pero describiendo con especial interés qué las convierten en algo tan popular no tanto para el usuario de a pie, sino para los miembros de la comunidad de desarrolladores de *software*.

2.1.1 Twitter

Comenzó como un proyecto de investigación en *Obvious Corporation*, una pequeña empresa de San Francisco. Inspirado por el modelo de mensajería SMS (*Short Message Service*) móvil a móvil, Jack Dorsey dio forma a una red social basada en el envío de mensajes informativos de hasta 140 caracteres. Con una audiencia inicial orientada a Estados Unidos, el 4 de Noviembre del año 2009 apareció la versión de Twitter en español.

Con una audiencia presente ya global y multi-idioma, Twitter ha ido perfilándose como una red social informativa de corte periodístico donde los usuarios se segmentan en una categoría joven-adulta, de corte cultural medio-alto, y cuyos intereses comunes de todos ellos recaen en compartir información, noticias y novedades, razones por las cuales los principales medios de prensa mundiales – televisión, radio y prensa escrita – apoyan y participan activamente en ella.

Twitter ofrece herramientas y librerías a disposición de desarrolladores de software que facilitan la integración de sus servicios y fomentan la investigación universitaria. Así, la autenticación de usuarios a través de su sistema, su API [1] de consumo y consulta, o proyectos propietarios como Storm [2] o Twitter Bootsrap [3] que han sido liberados como proyectos *open source*, convierten a Twitter en no sólo una red social, sino en una herramienta de integración social con otras aplicaciones, productos o proyectos de investigación, siendo considerada como punta de lanza tecnológica por muchos sectores la comunidad de desarrolladores de software.

2.1.1.1 Desarrollo de Apps

Cualquier usuario de Twitter puede crear su propia aplicación de Twitter. El concepto de aplicación se limita únicamente a crear unos credenciales de acceso para que cualquier aplicación de consumo externo pueda utilizar emplear los diferentes APIs de contenido con los que cuenta Twitter, que detallaremos en el siguiente punto.

Twitter, además, permite integrar algunas funcionalidades en desarrollos de terceros con el fin de expandirse y de ganar presencia incluso en software de terceros. Ofrece:

- **Botón de tweet**: permite twittear un contenido dado siempre que incluya este botón. Un ejemplo clásico de integración del botón de tweet podría ser un Portal Web de noticias, donde cada una de las noticias detalladas incluyen botones de redes sociales que permiten compartir dicho contenido en ellas.

- **Botón de seguimiento**: similar al anterior, solo que el usuario que pulse dicho botón seguirá en Twitter al usuario que estuviera relacionado con él. Este botón tiene gran éxito en, por ejemplo, blogs, pues permiten al lector seguir en Twitter rápidamente al fundador del blog, el autor de una determinada noticia, etc.

- **Twitter *timeline***: caja de contenido que incluye los últimos tweets de un determinado usuario o tema en tiempo real. Permite que portales Web, blogs, etc. puedan integrarlo.

- **Twitter *cards***: pequeños esquemas de contenido similares a tarjetas de visita que permiten mostrar contenido personal bajo ese mismo formato.

2.1.1.2 API

Twitter almacena diariamente cientos de millones de comentarios (*tweets*) vertidos por sus más de quinientos millones de usuarios activos en ella. Toda esta ingente y creciente actividad puede ser consumida a través de los diferentes APIs que ofrece a la comunidad de desarrolladores conceptualmente de dos modos muy distintos:

- **Búsquedas**: a través de llamadas a su API, podemos acceder a los diferentes tweets en base a nuestros criterios de búsqueda. Dado que la actividad de Twitter es vertiginosa, este método de búsqueda muy probablemente nunca obtenga en sus respuestas tweets recientes debido al alto coste que tendría tener una arquitectura en tiempo real en cada uno de los nodos que compone la arquitectura de Twitter.

  Este tipo de consumo de datos se hará a través de sus REST APIs, que a continuación detallaremos. Es importante recalcar que, si bien las búsquedas serán flexibles y se adaptarán a los diferentes criterios y filtros que elijamos, los resultados de búsqueda nunca representarán datos realmente inmediatos.

  Sin embargo, esto no siempre es una desventaja, pues en determinadas ocasiones – dependiendo de los requisitos o funcionalidades del software – querremos acceder a tweets antiguos para recuperar información pasada y analizarla en el presente.

- **Canal siempre abierto**: este segundo modo de consumo de datos es en tiempo real. La operativa consiste en establecer una comunicación con Twitter y mantenerla siempre abierta. Podremos elegir qué tipo de contenido filtrar como, por ejemplo, seguir determinados términos que puedan aparecer en los *tweets*, o definir un radio geográfico para recuperar *tweets* sólo de una determinada ciudad o zona.

  Desde el momento en que la conexión se crea, se establece un canal de comunicación permanente con Twitter (*stream*) y se recibirán tweets en tiempo real que se ajusten a nuestros criterios de filtrado.

  Esto, a diferencia del consumo de datos a través de búsquedas (REST APIs), permite tener acceso a *tweets* inmediatos. Sin embargo, nunca podremos acceder a tweets antiguos a la fecha en que creamos la vía de comunicación permanente con Twitter. A

este tipo de APIs se denominan *streaming* APIs, o APIs de consumo de datos en tiempo real.

Actualmente Twitter tiene tres tipos de APIs, dos de ellas son REST y una tercera es en *Streaming*:

- **REST API**: API de consulta y/o modificación de estados, permitiendo obtener *tweets* del hilo de contenido volcado por el usuario (*timeline)*, así como también añadir nuevo contenido (*tweets*) si la *App* tiene permisos de escritura.

- **Search API**: API de búsqueda que permite obtener *tweets* a partir de determinados términos de búsqueda tales como nombres de usuario, etiquetas (*hashtags*) o términos de búsqueda libre. Permite obtener resultados de un índice de contenido de hasta una semana de existencia. Esto impide que podamos buscar *tweets* más antiguos, así como tampoco *tweets* recién publicados. Este último punto es especialmente importante y en ocasiones lleva a confusión.

- **Streaming API**: API en *streaming* que, esta vez sí, permite recibir contenido en tiempo real. Este API se utiliza a menudo en desarrollo de aplicaciones que monitorizan Twitter con fines estadísticos o de análisis de impacto.

Para hacer uso de estas APIs Twitter utiliza una autenticación OAuth 1.0 [4] . OAuth es un protocolo de autenticación a tres bandas que permite autenticar o validar a un usuario en una aplicación, *Website*, etc. siempre que una tercera entidad dé su visto bueno. Ampliaremos con más detalle la información relativa a OAuth en una sección posterior.

2.1.2 Facebook

Facebook surgió en el año 2004 como un proyecto universitario de Mark Zuckerberg. Mientras cursaba sus estudios en la Universidad de Harvard desarrolló un prototipo que daba servicio a los estudiantes de su universidad donde cada usuario tenía su página recogiendo brevemente su información personal. El servicio se hizo popular y pronto se migró a otras universidades de Estados Unidos como la Universidad de Boston, el MIT y otras.

Tan sólo un año después Facebook logró alcanzar la cifra de un millón de usuarios registrados al expandir su audiencia a escuelas secundarias y a universidades extranjeras. El que fuera inicialmente un proyecto universitario pronto se consolidó como una empresa solvente y rentable al recibir una fuerte inversión de capital adicional.

Facebook incluye funcionalidades de intercambio de mensajes privados, conversación en tiempo real mediante chat, subida y almacenamiento permanente de imágenes y vídeo,

búsqueda y seguimiento de usuarios, *geolocalización*, creación de perfiles personales, profesionales o de grupo, aplicación Web y aplicaciones en dispositivos móviles etc.
Además, incluye un API de consumo externo, Facebook *Graph*, que permite la integración de la red social en aplicaciones y Portales Web de terceros: autenticación de usuarios a través de Facebook, inclusión de botones para compartir y comentar contenido externo en Facebook, etc.

Superando cualquier expectativa inicial, hoy cuenta con más de 800 millones de usuarios únicos registrados. Recientemente ha centrado su desarrollo de negocio en las plataformas móviles, donde ha empezado a explorar nuevos formatos publicitarios orientados a los gustos, aficiones y, en definitiva, al perfil del usuario propietario del móvil. Este nuevo movimiento de Facebook es, además, integrador, pues establece lazos con Google Play [5] y Apple iTunes Store [6] permitiendo publicitar Apps para móviles en Facebook y enlazando la descarga con sus respectivos *Markets* de aplicaciones.

2.1.2.1 Desarrollo de Apps

Facebook ha construido entorno a sí un ecosistema de desarrollo de software que permite a la comunidad de desarrolladores construir productos que hagan uso de alguno de los servicios que Facebook ofrece extendiendo su significado más allá de la acepción de red social.

La red social experimentó un mismo grado de crecimiento y de evolución que el llamado Boom de la Web 2.0 a principio del siglo XXI, de modo que su evolución a ecosistema de desarrollo partió desde la Web. Alguno de los servicios y utilidades que ofrece son:

- **Botón *'Like'*:** botón que accionaba un mecanismo de valoración positiva entorno a una idea, vídeo o, en definitiva, contenido. Lo que en la fecha presente supone una acción trivial y cotidiana, totalmente extendida en la Web, ayudó a muchos desarrolladores a mejorar su contenido en la Web y a ampliar su presencia en Facebook.

- **Botón 'share'**: paralelamente al botón 'like', esta otra acción permite compartir ese contenido asociado. Conjuntamente con el anterior botón, el botón 'share' ha permitido mejorar el posicionamiento de portales Webs redefiniendo el significado de Facebook como red social personal ampliándolo a un ámbito más genérico: ocio, negocios, opinión, etc.

- **Autenticación de usuarios:** mediante una implementación del protocolo OAuth 2.0 [7], Facebook provee los mecanismos necesarios para que terceros puedan certificar los credenciales de un usuario a la hora de llevar a cabo una acción clásica de 'login' en un portal Web. Esta acción, además, permite acceder a ciertos datos pertenecientes al usuario que lleva a cabo la autenticación, algo que forma parte del conjunto de técnicas de *Data Mining* más habituales hoy día.

A finales de la primera década del siglo XXI se experimentó el inicio de nueva revolución que hoy está en su máximo apogeo: la *Mobile Revolution*. Esta revolución apunta a la movilidad de contenidos de la Web acompañada de la aparición de teléfonos inteligentes con conectividad de datos que permiten el acceso a Internet.

Esta nueva brecha tecnológica ha dado lugar al crecimiento de grandes compañías como Apple, Google, Microsoft, Blackberry o recientemente Firefox, cada una de las cuales ofrecen soluciones de hardware y software distintas en temática de terminales móviles, pero siempre con un objetivo común: que el usuario esté continuamente conectado a la Red, consumiendo y volcando datos de manera constante.

Facebook no ha dado la espalda este avance, y buena parte del éxito en los últimos años se debe al apoyo en el sector móvil desde diferentes ángulos:

- **Perspectiva del desarrollador:** el soporte al desarrollador se ha ampliado, ofreciendo SDKs (*Software Development Kit*) de desarrollo para entornos Android (Google) y iOS (Apple) que permiten hacer uso de Facebook Graph [8] de una forma cómoda. Estas dos nuevas plataformas se suman al ecosistema Web ya existente. Además, ha apostado por la presencia de un mercado de juegos en la propia red social, ofreciendo la oportunidad a la comunidad desarrolladora a crear juegos que estén integrados en Facebook.

- **Perspectiva del consumidor:** Facebook apostó desde sus inicios en el entorno móvil, donde inicialmente ofreció gratuitamente aplicaciones móviles de su red social para la mayoría de los entornos mayoristas: Android e iOS. Además, el mes de Febrero del año 2013 Facebook anunció su propia versión instalable del sistema operativo Android. De esta manera, todos los servicios de Facebook dejan de integrarse sólo en una aplicación móvil para superar sus horizontes y formar parte total del propio software del terminal.

- **Perspectiva de negocio:** en Mayo del año 2012 Facebook salió a bolsa con unas expectativas antes nunca vistas. Sus acciones fueron en decremento, pero a día de hoy su valor permanece en alza. Es la primera vez en la Historia que una red social adquiere este estatus bursátil. En otro orden, el aumento de una audiencia cada vez más móvil, junto con la segmentación de población que per se ya definen los usuarios de Facebook (sexo, edad, población, etc.), ha potenciado el negocio de la publicidad móvil, aumentando sus ingresos.

2.1.2.2 API

Facebook ofrece varias de vías de integración con su sistema a través de un ecosistema de APIs que satisfacen desde diferentes enfoques las necesidades que pueda tener una Aplicación de Facebook en función de su objetivo.

A continuación se listan, junto con una breve descripción, los diferentes APIs disponibles:

- **Graph**: es, sin lugar a dudas, el API más popular y, según las estadísticas, el más utilizado por la comunidad de Desarrolladores. El Graph API es un HTTP [9] (*Hypertext Transfer Protoco*l) REST API que ofrece acceso al grafo social de Facebook a través de, como siempre, autenticación por OAuth 2.0

  A través del grafo social de Facebook podemos extraer la información relativa a los diferentes perfiles sociales de aquellas personas que accedan a nuestra Aplicación de Facebook a acceder a dicha información.

  Así, a través de este API podremos obtener todo el detalle del perfil de usuario, desde datos comunes como su nombre, apellidos o cuenta de correo, o algunos otros de especial interés de estudio como los grupos a los que sigue, sus comentarios en su muro o, incluso, las imágenes en las que está etiquetado.

  UniAffinity empleará este API para minar la información del usuario que posteriormente empleará para trazar parentescos entre usuarios del sistema.

- **FQL (Facebook Query Language)**: implementa un API basado en sintaxis de consulta SQL que permite acceder a los mismos datos que expone el Graph API, solo que esta vez emplea consultas HTTP con sintaxis SQL [10]. Esta sintaxis, de conocimiento mayoritario entre la comunidad de desarrolladores, hace que el acceso al grafo social de Facebook sea para muchos de ellos algo más sencillo.

- **Open Graph**: permite a Aplicaciones publicar nuevas acciones de un usuario en el grafo social de Facebook en relación con los siguientes subconjuntos de actividades:

  1. *Fitness*: publicación de estados de inicio o cese de actividades deportivas.

  2. *Música*: publicación de estados de reproducciones musicales, seguimientos de listas de música, etc.

  3. *Noticias*: publicación de estados de carácter periodístico tales como noticias periodísticas, noticias extraídas de blogs, etc.

  4. *Comentarios generales: likes* o seguimiento de grupos.
  5. *Vídeos:* publicación de estados de compartición de vídeos.

  6. *Libros:* publicación de estados que indican que el usuario ha leído un libro, etc.

- **Dialogs**: ventanas o diálogos Web que permiten integración en Portales Web y que autorizan al usuario visitante de, por ejemplo, el Portal Web que lo integre, hacer login a través de Facebook, enviar invitaciones de amistad o incluso publicar en el muro de un usuario.

- **Chat:** integración con el sistema conversacional de Facebook que permite integrar en tu propia Aplicación esta funcionalidad.

- **Localization and translation:** ofrece *geolocalización* para las Aplicaciones en base a sistema GPS si hablamos de aplicaciones nativas de dispositivos móviles, o geolocalización a través de posicionamiento IP para aplicaciones generales. Además, Facebook da soporte a cerca de 70 idiomas, ofreciendo un sistema de traducción para todas aquelals aplicaciones que deseen incluirlo.

- **Ads:** integración con el sistema de monetización publicitaria de Facebook, permitiendo gestionar, añadir, borrar, editar… campañas publicitarias en la red social.

- **Public feed:** muy relacionado con Graph API, permite acceder a todos los comentarios de los perfiles que hayan sido configurados como de acceso público. A diferencia del resto, este API no requiere autenticación, pues es público para cualquier usuario lector, incluso.

- **Keyword Insights:** a través de un término o palabra, Facebook expone el seguimiento o repercusión que tenga en su grafo social. Por ejemplo, si deseáramos saber cuáles son las ciudades en las que más veces algún usuario ha mencionado en sus publicaciones a "Obama", este API ofrecería esta clase de resultados fruto del análisis de los términos de todas las publicaciones.

Muy relacionado con los APIs anteriores, recientemente Facebook ha anunciado el lanzamiento de **Facebook Graph Search [11]**. Poco se sabe de este nuevo producto, pero el anuncio indica que ofrecerá un nuevo modelo de búsqueda mucho más intuitivo, práctico y social a través del grafo de conocimiento de Facebook que ofrezca respuestas a los usuarios similares a los siguientes ejemplos citados:

- *"Personas a las que le gusta el ciclismo"*.

- *"Fotos anteriores al año 1990"*.

- *"Fotos de Nueva York en las que salgan mis amigos"*.

- *"Canciones que me puedan gustar"*.

Su objetivo es muy similar al de *UniAffinity*: a través de su vasta base de conocimiento, ofrecer entidades, conceptos, personas… que puedan ser de interés para uno mismo.

Actualmente Facebook Graph Search se encuentra en fase beta en Estados Unidos. En el resto de continentes se ha abierto una lista de espera para participar en futuras versiones beta.

2.2 Protocolo OAuth

A mediados de la primera década del siglo XXI la Web experimentó un crecimiento enorme. Empujados por esta inercia, numerosos productos y servicios fueron apareciendo en la Red a los que los usuarios podrían acceder identificándose con unos credenciales que, a priori, deberían ser únicos e intransferibles para ese usuario y que le permiten acreditarse como él mismo, y no otro.

Al mismo tiempo, la inseguridad fue creciendo alimentada en cierta medida por una población que aún no había sentado cátedra en la Web y que eran objeto de malas prácticas en ella; principalmente, el uso prolongado de los mismos credenciales (usuario, contraseña) en diferentes Portales Web o servicios, abriendo las puertas a que cualquier usuario malintencionado que llegara a hacerse con estos datos pudiera acceder de una manera más o menos sencilla a datos comprometidos: cuentas bancarias, números de tarjeta de crédito, datos postales, etc. La seguridad en la Red necesitaba normas, protocolos, estándares a seguir.

El protocolo de seguridad OAuth versión 1.0, definido en la RFC 5849 [12], surge en Diciembre del año 2007 como alternativa de seguridad en la Web en desarrollos de software íntimamente ligados a APIs y otros servicios *SaaS* (*Software as a Service*) que se emplean HTTP como vía de comunicación entre cliente-servidor. Tras numerosas revisiones a lo largo de los años 2008 y 2009 su definición concluye en Abril del año 2010, convirtiéndose pronto en un referente de seguridad en la Web por su creciente aceptación.

Su punto fuerte radica en que en el proceso de autenticación de un usuario no intervienen de ningún modo los datos personales de dicho usuario tales como correo o contraseña. De este modo, y de forma totalmente estándar y universal para todos aquellos Portales Web o servicios que lo pongan en práctica, los usuarios no ven expuestas su intimidad en su día a día en la Red, reduciendo el riesgo de pérdida de identidad.
Esta primera versión del protocolo se inspiró en otros protocolos de seguridad propietarios como, por ejemplo, Google AuthSub [13], Yahoo BBAuth [14] o Flickr API [15]. Todos estos protocolos de seguridad daban una solución de autenticación donde los datos personales del usuario no se veían comprometidos, pero siempre muy orientada a la Web.

Una de las grandes novedades que arrojó a la luz OAuth fue, junto con la consabida versión universal de seguridad nacida fruto de la experiencia de fuentes como las citadas, una versión

de autenticación orientada a aplicaciones de sobremesa, dispositivos móviles y centros multimedia de reproducción de audio y vídeo conectados a Internet.

2.2.1 Roles

El protocolo OAuth 1.0 define fundamentalmente tres tipos de roles que intervienen en el proceso de comunicación y autenticación.

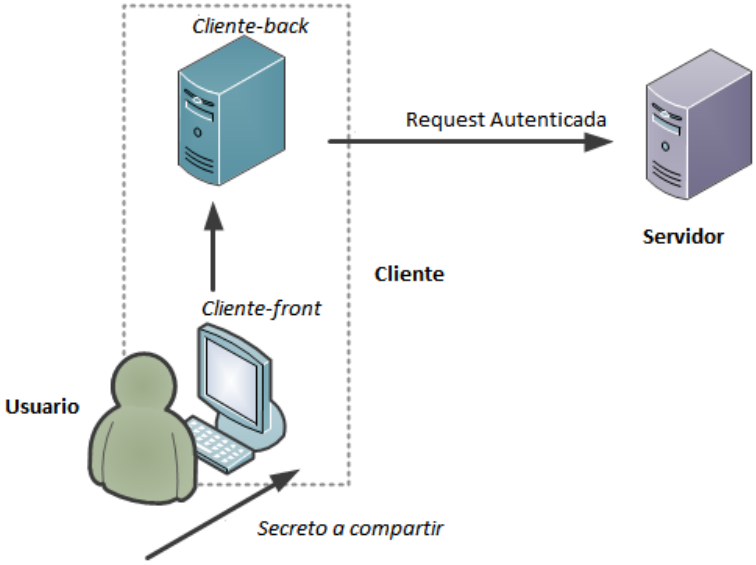

Figura 1: roles intervinientes en OAuth 1.0

Cada uno de estos roles se definen como:

- **Cliente**: se define como aquella parcela de software que da acceso a cierto contenido asociado al usuario o propietario del contenido. Por citar un ejemplo ilustrativo, un cliente podría ser un Portal Web como Facebook. En determinados tipos de cliente podemos encontrar, a su vez, una segunda subdivisión totalmente opcional:

    - Cliente-front: se trata de la aplicación cliente propiamente dicha. Esto es, y continuando con el ilustrativo ejemplo de Facebook, se trataría de la Aplicación Web en sí misma, únicamente.

    - Cliente-back: se consideran componentes de *backend*, pero enmarcados en el rol de cliente, todos aquellos servidores que interactúen de alguna manera

con la parte de cliente-front. En el caso de Facebook probablemente se cuente con componentes de *backend* que se comunican con el cliente-front para generar notificaciones o alertas, otros servidores pueden facilitar comentarios asociados a una imagen para que el cliente-front los muestre, etc.

El cliente-back es, en definitiva, ese ecosistema de aplicaciones servidoras que nutren al cliente en su totalidad. A su vez, alguno de estos componentes será el que típicamente se comunique con el componente que ejerza el rol de Servidor, evitando así que sea el Usuario el que se comunique directamente con el Servidor creando una comunicación a tres bandas.

- **Servidor**: componente que acredita o desestima el acceso a un determinado recurso. Es la entidad que garantiza o deniega accesos a recursos de un usuario.

- **Usuario o propietario del contenido**: persona física que desea acceder a un recurso inherente a sí mismo. En el caso de Facebook, el recurso al que el usuario podría acceder sería toda aquella información que haya volcado en la red social (fotografías, comentarios personales y de amigos, etc.) que sólo puede obtener si se autentica.

2.2.2 Credenciales y *tokens*

OAuth usa fundamentalmente dos tipos de credenciales: credenciales de cliente (*consumer key* y *secret*) y los llamados credenciales de *token* (*access token* y *secret*). Aunque también se contemplan un tercer tipo de credenciales llamados credenciales temporales, en la práctica no se suelen emplear.

Los **credenciales de cliente** permiten al cliente autenticarse. Esto permite al servidor a en recoger información sobre el cliente usando sus servicios, ofrecer a algunos clientes algún trato especial, o proveer al usuario y dueño del contenido algún tipo de aviso sobre si el recurso que al que requiere acceso (ver una imagen, por ejemplo) está protegido o no.
Los **credenciales de token** son aquellos asociados al Usuario o dueño de los recursos que se emplean en lugar de los clásicos usuario y contraseña y que evitan que esta información comprometida sea divulgada. El Cliente emplea estos credenciales de *token* para acceder a los recursos del Usuario.

Estos credenciales de *token* son generados aleatoria y unívocamente, y se componen de un identificador de *token* y en ocasiones - la mayoría - de una cadena alfanumérica que es única y que está generada a partir del *token secret*. Para generar esta cadena alfanumérica se suele seguir esta buena práctica:

1 Primero generamos una cadena de texto alfanumérica totalmente aleatoria. En función del lenguaje de programación que estemos empleando hay diferentes maneras de obtener este tipo de cadenas aleatorias, que siempre suelen estar relacionadas con la marca de tiempo o *timestamp* actual.

2 Mediante HMAC [16] (*Hash based Message Authentication Code*) generamos a partir de ese *secret* la cadena alfanumérica aleatoria y única y un algoritmo de codificación a elegir (por ejemplo, SHA256 [17]) un texto cifrado.

En Aplicaciones Web es frecuente que este valor cifrado que hemos generado a partir del *secret* se codifique en Base64 [18] y sea inyectado en una Cookie asociada a cierto dominio y a cierta duración máxima.

2.2.3 Protocolo

Pero el protocolo OAuth no se emplea únicamente para autorizar acceso a un usuario a un determinado recurso, sino también como un único sistema de autenticación. La diferencia entre autenticar y autorizar se halla en que el proceso de autenticación únicamente valida al usuario como un usuario que está dentro o fuera del sistema, mientras que el proceso de autorización otorga privilegios al usuario para acceder a un determinado conjunto de recursos que de los que dispone el sistema.

En la práctica, OAuth se emplea en un primer plano para autenticar, y posteriormente para autorizar al usuario. Para una mayor comprensión del protocolo, los roles que intervienen y algunas peculiaridades del protocolo, se detalla un ejemplo ilustrativo con Twitter que servirá de antesala para próxima sección del presente documento técnico.

Supongamos que tenemos un Portal Web que permite a los usuarios hacer *login* en él. Este Portal Web ejercerá el Rol de Cliente, y nosotros el rol de Usuario. La acción de *login* se hará por OAuth y emplearemos Twitter para dar validez a los credenciales de usuario, de modo que si un usuario tiene cuenta en Twitter y hace *login* podrá, por ejemplo, acceder a una sección de nuestro Portal Web que quede restringida para usuarios anónimos.

Veamos a continuación el flujo de comunicación aplicado a nuestro caso y cómo interactúan los diferentes componentes:

1. El Cliente pide un *token* al Servidor. Esta operación es transparente para el Usuario.

2. El Cliente redirige al usuario a una página segura en el Servidor, pasándole el credencial *token* como parámetro.

3. Este credencial de *token* se adjunta de dos maneras distintas: por un lado, por una *Cookie*; por otro, a través de un parámetro de la URL (*Uniform Resource Locator*) que nos lleva al formulario de *login* de Twitter.

4. Esta página a la que se le redirige queda bajo el dominio del Servidor, y se trata del formulario de *login* de Twitter.

5. El Usuario se autentica en la página del Servidor, validando así ese credencial de *token* que se había adjuntado de manera transparente. De este modo el Servidor recoge el credencial de *token* para que una vez el usuario ha hecho *login* con éxito el Servidor pueda autenticar ese credencial de *token*.

6. El Servidor envía al usuario de vuelta a la página del Cliente especificada mediante un parámetro opcional llamado `oauth_callback`. Este parámetro define a qué página del Cliente (por ejemplo, de nuestro Portal Web) debe redirigirse tras una operación de autenticación exitosa. Del mismo modo puede facilitarse una *oauth_callback_error* para redirigir a una página de error dentro del ámbito del Cliente.

7. El Cliente recoge el credencial de *token* del Usuario ya autenticado desde el lado Servidor.

8. El Cliente puede pedir recursos al Servidor para los que el Usuario esté autorizado pasando siempre en cada petición HTTP ese credencial de *token* que ha autenticado. Estos recursos pueden ser, por ejemplo, el nombre de la cuenta de usuario de Twitter del Usuario, sus últimos tweets publicados, el número de *followers* que tiene, etc.

Este tipo de comunicación es cada vez más común no sólo en aplicaciones Web, sino también en otro tipo de aplicaciones de sobremesa o móviles.

2.2.4 OAuth 2.0

Aunque desde el año 2009 ya se empezó a definir una nueva versión de OAuth, a día de hoy no existe una definición completa del nuevo protocolo, y de hecho no hay una fecha estimada para

su finalización, por lo que su análisis no se contempla en este documento técnico debido al aspecto aún cambiante de su contenido.

El carácter no oficial y mutable de OAuth 2.0 no ha impedido que grandes compañías como Facebook implementen alguno de sus borradores. En concreto, el protocolo de autenticación y autorización de OAuth 2.0 que Facebook ha implementado corresponde al borrador número doce de OAuth 2.0.

2.3 Apache Solr

El proyecto Apache Solr [19] fue creado inicialmente por Yonik Seeley en el año 2004, quien por esa fecha trabajaba para la empresa CNET Networks. Solr fue desarrollado como un producto que ofrecía funcionalidades de búsqueda para el portal web de la compañía al estilo del popular buscador de Google.

En el año 2006 la compañía CNET Networks decidió liberar el código, enteramente desarrollado con lenguaje Java, y ofrecerlo a la Fundación Apache (*Apache Foundation* [20]) donde, al igual que cualquier otro proyecto liberado, y siguiendo las pautas o protocolo de trabajo que mantienen, pasó un tiempo en estado de incubación que permitió organizar el proyecto Apache Solr desde el ámbito legal y financiero.

Las funcionalidades de Solr son diversas, pero se define a sí mismo como un motor de búsqueda documental en el que se habilitan mecanismos para añadir o indexar nuevos datos (documentos) sobre los que elaborar consultas de búsqueda posteriores. En el tratamiento de los documentos, cuyo contenido es puramente textual, Solr habilita diferentes mecanismos de tratamiento de texto.

Actualmente Apache Solr es uno de los proyectos más activos de la Fundación Apache, habiendo alcanzado la versión 4.3.0, dato que corrobora la madurez del proyecto. Cuenta con una comunidad muy activa de desarrolladores y de colaboradores varios, además de varias listas de correos con actividad diaria.

2.3.1 Arquitectura

Solr se estructura principalmente en dos capas bien diferenciadas: un lado servidor y un lado cliente.

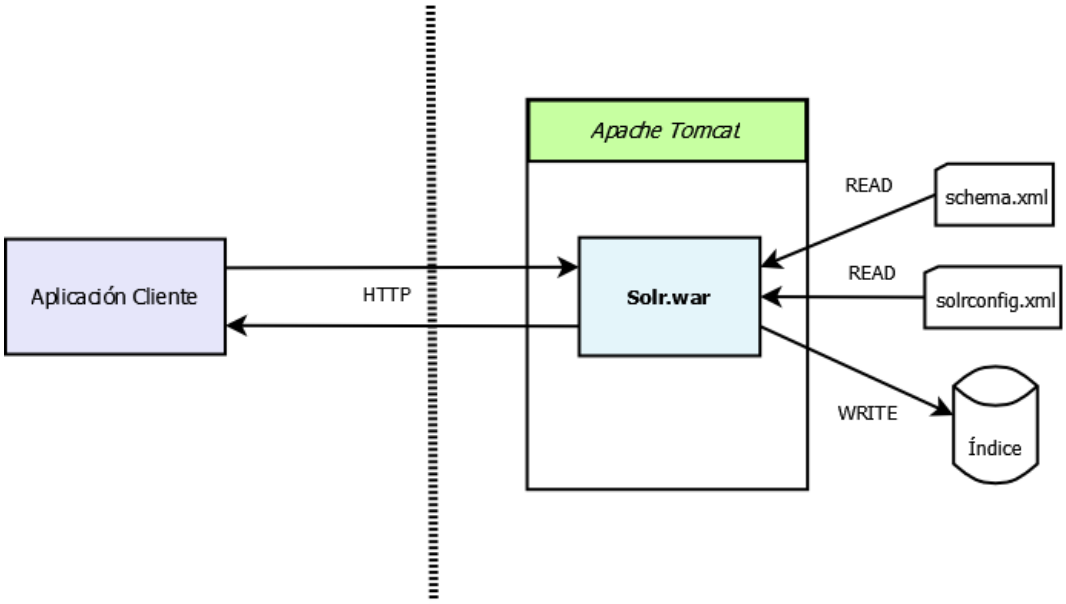

Figura 2: arquitectura de Apache Solr

- **Capa servidora:** Solr se distribuye empaquetado en un archivo de extensión .war que permite desplegarse en un servidor de aplicaciones como, por ejemplo, Apache Tomcat [21] o Jetty [22]. Una vez se ha desplegado correctamente, Solr habilita una REST API que permite lanzar consultas o añadir nuevos documentos al índice del servidor desplegado desde la capa Cliente.

  Para que un servidor Solr pueda conocer la ruta en la que debe almacenar los datos para el índice, la naturaleza de los datos, los tipos de campos a almacenar, así como otros parámetros que definen la configuración del propio servidor Solr, la capa Servidora cuenta con algunos archivos de configuración que especifican tales como `schema.xml`, donde se definen la naturaleza de los campos a almacenar, o `solrconfig.xml`, donde se configuran el resto de parámetros asociados a un servidor Solr.

- **Capa cliente:** para que cualquier desarrollo de software pueda comunicarse con la capa servidora de Solr existen numerosos conectores o *drivers* que facilitan esta comunicación entre Cliente y Servidor. El más popular de todos ellos, que se incluye también en cada nueva versión de Apache Solr, es SolrJ [23], el driver nativo de lenguaje Java.

  Sin embargo, existen muchos otros conectores nativos disponibles para otros lenguajes de programación tales como PHP o Python. Este amplio espectro de conectores se debe, en parte, a que el lado Servidor de Solr expone una REST API, lo que facilita en gran medida la integración desde el Cliente al tratarse únicamente de peticiones HTTP lo que se lanzan contra el servidor desplegado.

Apache Solr admite, además, diferentes configuraciones de instalación:

- **Servidor standalone:** un único servidor Solr desplegado en un servidor de aplicaciones. Por defecto, un servidor Solr realizará operaciones de búsqueda e indexaciones sobre un único índice. Sin embargo, un servidor Solr puede contar con uno o varios subconjuntos de datos denominados 'cores'. Cada core irá asociado a un único índice de datos.

- **Configuración maestro-esclavo:** un servidor Solr hará las veces de maestro y podrá tener varios servidores Solr configurados como esclavos. Cada uno de los servidores puede estar en entornos distintos. Por formalidad y buenas prácticas, las operaciones de escritura (indexación) se harán contra el servidor maestro, mientras que las búsquedas podrá hacerse contra cualquiera de los servidores esclavos, pues mantendrán sincronía con los datos que el servidor maestro tenga gracias a un sistema de replicación de documentos entre maestro-esclavos con el que Apache Solr cuenta.

- **SolrCloud**: nuevo sistema de distribución de índices incluido desde la versión Apache Solr 4.0. Permite configurar una topología de servidores Solr en la que cada uno de ellos tendrá en su índice de datos un tanto por ciento del total de los datos indexados, permitiendo establecer operaciones de búsqueda e indexación distribuidas de un modo totalmente transparente desde el punto de vista del cliente que interactúe con SolrCloud. Además, integra una configuración maestro-esclavo entre los diferentes nodos que componen la topología aportando, además de la escalabilidad horizontal, una tolerancia a fallos robusta y completa.

2.3.2 Apache Lucene

No podríamos comprender Apache Solr sin hacer una breve incursión sobre Apache Lucene [24], pues Lucene es el propio núcleo o *core* de Solr y se trata también de un proyecto *opensource* que forma parte de la Fundación Apache.

Lucene es una tecnología para la Recuperación de Información, o Information Retrieval, que realiza operaciones de indexación y búsqueda sobre los datos previamente indexados. Cuenta con un API desarrollado en Java que habilita el procesado de documentos con formatos `.txt`, `.pdf`, `.doc`, `.xml`, `.html` y otros. A partir de la lectura de dichos documentos, Lucene ofrece una serie de funcionalidades que permiten desgranar la información vertida en ellos, adaptarla a nuestras necesidades, cribándola, para finalmente ser volcada en un índice de datos.

Una de las características más destacable de Lucene es que incluye un algoritmo de relevancia de resultados de búsqueda, basado en el algoritmo *Tf-Idf [25]*, que permite ordenar los resultados de una búsqueda teniendo en cuenta la relevancia de los términos que estamos buscando en el contenido de los documentos que se han indexado previamente y sobre los que lanzamos la consulta.

Este algoritmo de relevancia, similar en concepto - que no en ejecución - al famoso algoritmo PageRank [26] del motor de búsqueda de Google, junto con el listado de analizadores de texto que incluye Lucene, dan la posibilidad al desarrollador a construir un motor de búsqueda personalizado.

A diferencia de Solr, Lucene únicamente incluye las piezas o módulos necesarios - o útiles - para el desarrollo de un motor de búsqueda, pero no habilita ningún REST API que permita acceder a estas funcionalidades de búsqueda e indexación de una manera más sencilla o cómoda desde el punto de vista del desarrollador.

La gran diferencia entre Solr y Lucene - que también sirve de definición para ambos - es que Lucene facilita las herramientas de trabajo para el desarrollo de un motor de búsqueda, mientras que Solr es en sí mismo una solución de motor de búsqueda ya desarrollada que facilita buena parte del trabajo para el desarrollador de software.

2.3.3 Algoritmo de relevancia Tf-Idf

El algoritmo de relevancia *Tf-Idf*, o frecuencia inversa de un término en un documento, define la importancia o relevancia de una palabra o término en un conjunto de texto. En la práctica, este algoritmo mide la relevancia de una palabra en un documento de texto, algo que permitirá ordenar resultados de búsqueda en Apache Solr en base a los términos de búsqueda que hayamos indicado.

La frecuencia inversa implica un cálculo extremadamente sencillo. Así, si en un documento tenemos un término repetido N veces, la frecuencia inversa no será sino $1/N$. En un contexto no matemático esto implica que cuanto más frecuente sea un término en un documento menos relevante será desde un punto de vista objetivo y analítico. Por el contrario, si ese mismo término apareciera en un documento distinto en lugar de N veces $N-2$ veces, diríamos que es un término mucho más relevante en el documento número dos.

En Apache Solr se emplea el algoritmo Tf-Idf como base para el cálculo de la relevancia de un documento y así poder ordenar los resultados de búsqueda. A la hora de calcular la relevancia se tiene en cuenta los términos de búsqueda que indicamos, cuyos *Tf-Idf* tiene pre-calculados Solr y que actualiza con cada documento que se indexa.

$$Score(q,d) = \sum_{t \in q}^{T} tf - idf(t,d)$$

Figura 3: cálculo de la relevancia por Tf-Idf

Así, por cada documento en los que aparezcan los términos de búsqueda se calcula el sumatorio de los *Tf-idf* de los términos en dicho documento. El total es el resultado o *score* del documento obtenido para esa búsqueda. Por defecto, Solr retorna los resultados de búsqueda con una ordenación por relevancia, de modo que el documento con mayor resultado será el que aparezca en la primera posición, y así consecutivamente.

2.4 Bases de datos

2.4.1 Bases de datos relacionales

A finales de los años setenta, Edgar Frank Codd – antiguo trabajador de IBM – sentó las bases del modelo de entidad-relación (ER) que fijaba un nuevo paradigma en la agrupación y ordenación de datos que hoy nos es tan conocido. Este nuevo modelo plantea un esquema de organización con tablas para agrupar unos y otros datos en entidades lógicas, donde cada una de estas entidades será definida por un esquema o tabla conteniendo, a su vez, atributos que les caractericen.

Para cada una de estas entidades tendremos una tabla asociada en la base de datos no relacional, permitiéndose relaciones entre entidades a través de sus tablas. Así, si pensáramos en la entidad “perro”, tendremos una tabla en nuestra base de datos para esa misma entidad. Extendiendo dicha entidad para asociarle atributos, tendríamos, por ejemplo, los siguientes: raza, color de pelo, género, peso, etc.

Si deseamos añadir un nuevo registro (perro) a nuestra tabla, cada uno de estos registros serán filas en ella, siendo cada uno de los atributos de la misma las características definidas anteriormente. Cada una de esas filas o entradas a esa tabla serán todos y cada uno de los datos que tenemos para organizar.

Basándonos en este paradigma o premisa, podríamos tener una tabla en nuestra base de datos muy similar a la siguiente:

Entrada nº	Raza	Género	Color	Peso	Edad
0	Pastor alemán	Macho	Negro y marrón	12 kgs	1 año
1	San bernardo	Macho	Marrón	21 kgs	1 año
2	Caniche	Hembra	Negro	6 kgs	6 meses

Tabla 1: ejemplo de tabla ER

La complejidad del esquema ER es directamente proporcional a la complejidad de los datos a organizar y las relaciones que existan entre sí. El caso de ejemplo es extremadamente sencillo, pero en entornos complejos donde existan muchas tablas y relaciones entre ellas convierten a la base de datos en un sistema de comprensión y mantenimiento difíciles.

Desde los años setenta en adelante, este modelo de almacenamiento contó con la amplia aceptación del sector, lo que propició su mejora y desarrollo hasta hoy. Gracias a esa amplia aceptación del modelo nació el lenguaje SQL (*Structured Query Language*) como lenguaje estándar declarativo que permite acceder para insertar o consumir datos de este tipo de bases de datos relacionales, permitiendo especificar diversos tipos de operación sobre éstas: crear nuevas tablas, borrar nuevas tablas, introducir datos en una nueva tabla, actualizar datos en ellas, obtener registros de las tablas a partir de diversos filtros de consulta, ordenar registros para una consulta de búsqueda dada, etc.

A día de hoy existen muchas alternativas de bases de datos no relacionales en el mercado, siendo una amplia mayoría de todas ellas gratuitas. A destacar, MySQL [27], SQLite [28], PostgreSQL [29].

2.4.2 Bases de datos no relacionales

Aproximadamente el 95% de la información generada y volcada en la Red es información no estructurada; es decir, sin esquema de datos predefinido. Las nuevas tendencias tecnológicas apuntan a desarrollos de Software que, de un modo u otro, consumen esta información no estructurada de la Red para el desarrollo de aplicaciones que procesen dicha información.

El procesamiento de esta información debe enfrentarse a tres problemas fundamentales:

- No existe esquema, por lo que la base de datos que almacene los datos debe ofrecer alguna solución que ofrezca persistencia sin esquema.

- El volumen de datos es, en la mayoría de los desarrollos de software de este tipo, inmensos y crecientes en tamaño.

- Las labores de inserción de datos y de consulta de datos deben ser lo más rápidas posibles. La latencia de escritura y lectura es importante, pues muchos de estos desarrollos de software tienden a ser NRT (*Near Real Time*).

A raíz de estas tres necesidades fundamentales y contemporáneas, nuevos modelos de bases de datos, conocidos como bases de datos no relacionales o NoSQL (*Not Only SQL*), tratan de dar solución a las mismas desde enfoques diferenciados, pero con dos características comunes en la mayoría de ellas:

- Ofrecer opciones de configuración de la base de datos que ayude a escalar horizontalmente, apostando por la distribución de datos en N instancias o máquinas comportándose como una caja negra en la que, desde fuera, la configuración es tratada como una sola instancia de la base de datos fuera de esa caja negra.

- No aseguran el principio ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad):

    - Atomicidad: es la propiedad que asegura que la operación se ha realizado o no, y por lo tanto ante un fallo del sistema no puede quedar a medias.

    - Consistencia: Integridad. Es la propiedad que asegura que sólo se empieza aquello que se puede acabar. Por lo tanto se ejecutan aquellas operaciones que no van a romper las reglas y directrices de integridad de la base de datos. La propiedad de consistencia sostiene que cualquier transacción llevará a la base de datos desde un estado válido a otro también válido.

    - Aislamiento: es la propiedad que asegura que una operación no puede afectar a otras. Esto asegura que la realización de dos transacciones sobre la misma información sean independientes y no generen ningún tipo de error.

    - Durabilidad: es la propiedad que asegura que una vez realizada la operación, ésta persistirá y no se podrá deshacer aunque falle el sistema.

Podemos dividir las bases de datos relacionales en tres modelos de persistencia distintos:

1 **Bases de datos clave-valor:** orientadas al almacenamiento de datos asociados a una clave. Estos datos no son estructurados, y pueden ser datos numéricos, caracteres, cadenas de

caracteres, o composiciones algo más complejas como JSON. Dos de las bases de datos NoSQL clave-valor son Cassandra [30] y Redis [31].

2 **Bases de datos documentales:** orientadas al almacenamiento de documentos sin estructura, típicamente con formato JSON. Dos de las bases de datos NoSQL documentales más populares son MongoDB [32] y CouchDB [33].

3 **Bases de datos orientadas a grafos:** orientadas al almacenamiento complejo de datos con estructura de grafo incluyendo los vértices existentes entre sus nodos. La base de datos NOSQL orientada a grafos más popular es Neo4J [34].

2.4.3 MongoDB

Cuando DoubleClick, una de las primeras empresas proveedoras de publicidad online en la Web, fue comprada por Google, fueron varias las *startups* que surgieron a raíz de esa transacción económica. Una de ellas fue 10gen, creada por Dwight Merriman, el que fuera fundador y CTO de la propia DoubleClick, a finales del año 2007.

10gen está centrada en ofrecer productos y servicios que dieran soluciones tecnológicas a retos y problemas relacionados con la escalabilidad de los sistemas en el curso del almacenamiento y la recuperación de datos de un modo rápido y eficaz.

Por un lado, observaron que las bases de datos relacionales clásicas resultaban demasiado rígidas por necesitar un esquema definido de datos que respetar. Los datos provenientes de los intercambios de información durante la inyección de publicidad online son diversos, y no siempre todos son incluidos en las peticiones y en las respuestas de publicidad. Así, datos como el tipo del navegador, las coordenadas geográficas del usuario, dominio, lenguaje, etc., son datos esperables, pero no siempre imprescindibles.

Pero el grado de aporte tecnológico de 10gen no puede medirse sin conocer con detalle las diferencias entre bases de datos relacionales y bases de datos no relacionales. En los dos siguientes puntos se describen las peculiaridades de ambos modelos de datos.

La naturaleza mutable del contenido en ocasiones hace no respetar ningún esquema definido para un tipo de datos. Es ésa naturaleza cambiante del contenido, junto con la necesidad de persistir los datos de una manera distribuida en diferentes máquinas de una manera cómoda y ágil, la que alentó a los desarrolladores de DoubleClick a buscar una solución práctica a este reto tecnológico.

Además, y dado que su negocio estaba íntimamente ligado a la Web, vieron en Javascript [35], un lenguaje de amplio uso en el desarrollo Web, una solución práctica como lenguaje de desarrollo aplicado a este nuevo *framework*. Como resultado, nació MongoDB, una base de

datos no relacional o NoSQL desarrollada íntegramente con lenguaje C en su núcleo, pero cuyo lenguaje de desarrollo, inserción de nuevos elementos, consulta de los mismos, etc. es Javascript.

A diferencia de las bases de datos relacionales, muchas bases de datos NoSQL no tienen un esquema definido de datos, como es el caso de MongoDB, de modo que en un mismo conjunto de datos, que en nuestro ejemplo anterior bien podría ser el concepto *"perro"*, existirían distintas entradas de datos que no tienen por qué tener el mismo número de atributos entre sí.

En MongoDB los conjuntos de datos no están almacenados en tablas, sino en colecciones. Estas colecciones pueden ser creadas manualmente o automáticamente. MongoDB es muy flexible, de modo que si nos viéramos obligados a guardar un registro en una colección no existente el sistema se encargaría de crearla previamente para posteriormente hacer el guardado de estos datos entrantes.

A diferencia de las bases de datos no relacionales clásicas, MongoDB almacena el contenido utilizando el formato llamado BSON [36] (*Binary JSON*), que proviene de la estructura de datos empleada en Javascript llamada JSON [37] (*JavaScript Object Notation*). En esencia, un objeto BSON es un objeto JSON del que se obtiene su codificación binaria para que sea finalmente almacenada.

Un objeto JSON o BSON se compone de un listado de elementos relacionados entre sí por una clave y un valor, donde la clave es alfanumérica y el valor adopta diferentes formatos. Los valores pueden ser numéricos, alfanuméricos, listas o *Arrays* de elementos, fechas, o incluso pueden ser otros BSON. Una de las cualidades más destacables de este formato es su aspecto visual legible.

```
{
        "name" : "Luis",
        "university" : "Carlos III - Leganés",
        "age" : 28,
        "birthdate" : ISODate("2013-05-19T12:49:47.290Z")
}
```

Código 1: ejemplo de MongoDB BSON

MongoDB destaca por su configuración altamente escalable, admitiendo fundamentalmente tres tipos de configuración distintas:

- **Standalone:** una sola instancia de MongoDB en una única máquina. En esta única máquina será donde lancemos las consultas de búsqueda, inserciones de nuevos datos, etc.

- **Replica-set:** configuración de varias instancias o nodos de MongoDB. En esta configuración se define uno de los nodos como maestro, mientras que el resto de los nodos serán réplicas o esclavos. Aunque MongoDB permite inserción y búsquedas en todos los nodos, es recomendable que la práctica habitual sea que las operaciones de escritura se lleven a cabo en el nodo designado como maestro, mientras que las búsquedas se lancen contra los nodos configurados como esclavos.

  Los nodos esclavos guardarán sincronía con los datos que almacene el nodo maestro, de modo que si añadimos un nuevo BSON en el nodo maestro automáticamente se replicará esta información a todos los nodos esclavos que disponga nuestra arquitectura.

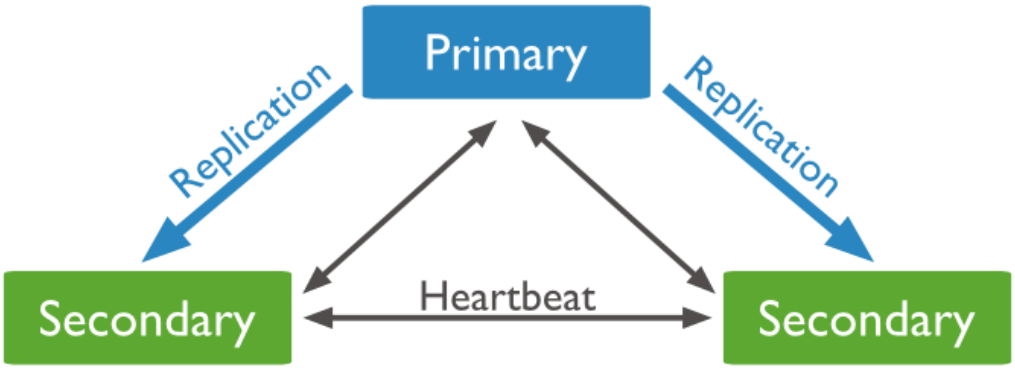

Figura 4: configuración básica MongoDB replica-set

Como vemos en la imagen, en una topología MongoDB replica-set contaremos con una instancia de MongoDB primaria o maestra junto con N instancias de MongoDB esclavas.

Las escrituras se harán siempre en el nodo maestro, mientras que los nodos esclavos se sincronizarán con el maestro-árbitro para tener actualizados los datos. Que tengan los datos actualizados es importante, puesto que las operaciones de lectura se harán contra los nodos esclavos.

En MongoDB replica-set cualquier nodo puede ser maestro y esclavo, pero nunca al mismo tiempo ambas. Es decir, el rol de maestro va rotando entre las instancias disponibles en la topología replica-set.

Existe un tercer rol de nodo llamado **árbitro** que es opcional, pero recomendable. Un nodo árbitro mediará en la elección de quién es el nodo maestro y quiénes son los esclavos. Son instancias de MongoDB que no almacenan datos, únicamente se encargan de decidir los papeles de cada actor disponible en la topología replica-set.

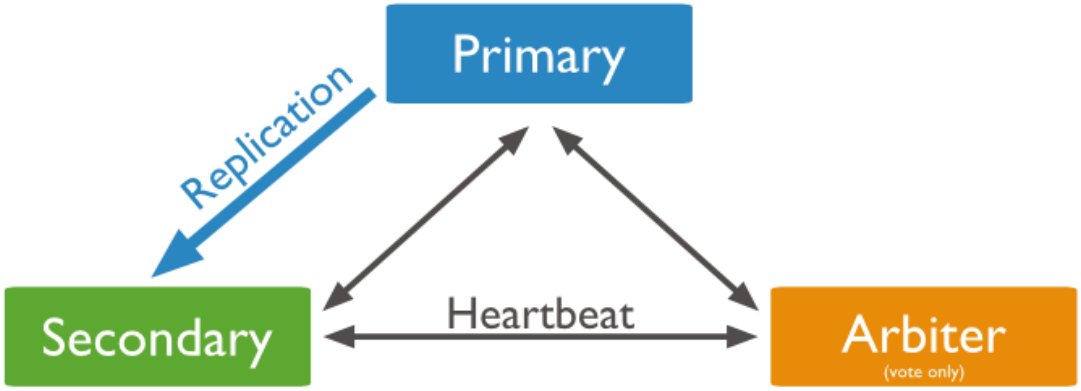

Figura 5: configuración MongoDB replica-set con árbitro

- **Cluster**: de las tres disponibles, ésta es la configuración más escalable horizontalmente de todas ellas. Permite la distribución de los datos en diferentes nodos en pequeños subconjuntos o *shards*, de modo que MongoDB actúa como una caja negra que vista desde el exterior actúa como una única instancia o nodo de MongoDB pero en realidad el *cluster* dispondrá de N instancias de MongoDB alojando todos estos datos.

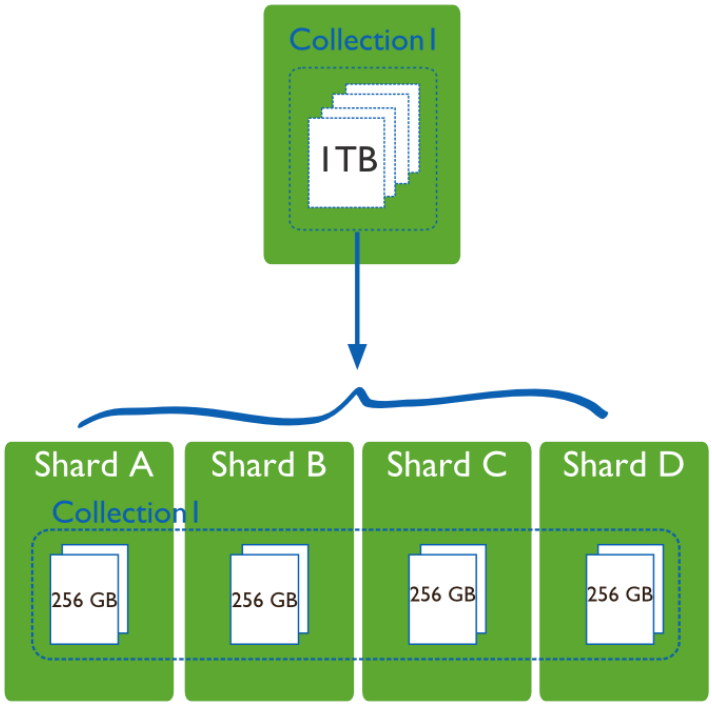

Figura 6: configuración básica MongoDB cluster

De este modo, una colección de datos en MongoDB será distribuido en tantos subconjuntos como instancias de MongoDB existan en la topología, procurando mantener cierto equilibrio a la hora de balancear el tamaño de los subconjuntos en todas las instancias, procurando que todas ellas tengan un tanto por ciento similar entre sí.

Este tipo de configuración en *cluster* admite, a su vez, réplicas de los nodos que lo componen, comportándose como réplicas de respaldo ante caídas. Así, si uno de los nodos del cluster que tenían el 20% de los datos deja de estar disponible por la caída de un servidor, su nodo de respaldo, que tendrá los mismos 20% de los datos, actuará como él. Si el nodo inhabilitado volviera a estar disponible, éste pasaría a ser el nodo de respaldo en su lugar.

MongoDB se ha extendido en los últimos años gradualmente, en parte por sus virtudes, en parte por el amplio abanico de conectores o *drivers* que permiten su uso en una gran cantidad de lenguajes de programación disponibles. Entre ellos, encontramos drivers para los siguientes lenguajes, *frameworks* y entornos de programación:

1. C, C++ y C#
2. Javascript
3. Java
4. Erlang
5. Node.js
6. Perl
7. PHP
8. Python
9. Ruby
10. Scala

2.5 Spring Framework

Spring Framework [38] ofrece un abanico de soluciones tecnológicas en lenguaje Java para el desarrollo de aplicaciones y módulos orientados a la Web, ofreciendo todos los recursos y herramientas que habitualmente son necesarios para este tipo de desarrollos, desde conectores con todo tipo de base de datos (MongoDB incluida), gestión de transacciones o la implementación del paradigma MVC (Modelo, Vista, Controlador) incluyendo *renderizado* de plantillas o *templates* en formato HTML, etc.

En el año 2001 los modelos dominantes de programación para aplicaciones basadas en web eran ofrecidas por el API Java Servlet y los Enterprise JavaBeans [39], ambas especificaciones creadas por Sun Microsystems, empresa creadora del lenguaje de programación que

posteriormente fue adquirida por Oracle, en colaboración con otros distribuidores y partes interesadas que disfrutaban de gran popularidad en la comunidad Java.

Los primeros componentes de lo que se ha convertido en Spring Framework fueron escritos por Rod Johnson en el año 2000 mientras trabajaba como consultor independiente para sus clientes en la industria financiera en Londres. Rod amplió su código para sintetizar su visión acerca de cómo las aplicaciones que trabajan con varias partes de la plataforma J2EE [40] podían llegar a ser más simples y más consistentes que aquellas que los desarrolladores y compañías estaban usando por aquel entonces.

Se formó un pequeño equipo de desarrolladores que esperaba trabajar en extender el *framework* y un proyecto fue creado en Sourceforge [41], comunidad dedicada a la proyección de proyectos *opensource*, en Febrero de 2003. Después de trabajar en su desarrollo durante más de un año lanzaron una primera versión en Marzo de 2004.

Poco después de este lanzamiento, Spring ganó mucha popularidad en la comunidad Java debido en parte al uso extensivo de la herramienta de documentación Java, Javadoc, disponiendo, además, de una amplia documentación complementaria muy detallada, algo poco habitual en un proyecto *opensource* y que fue recibido de buen grado por parte de la comunidad de desarrolladores.

A día de hoy el proyecto se encuentra maduro y estable, con una versión 3.2.0 disponible para la comunidad desarrolladora. Aun hoy, Spring continúa siendo un referente, y se trata de una de las soluciones técnicas más empleadas por los desarrolladores Java durante el desarrollo de módulos de negocio que tienen REST APIs y que necesitan algún tipo de acceso a base de datos, sea cual sea su naturaleza: relacional o no.

Este *framework* tan popular incluye muchos recursos útiles a disposición del programador, como por ejemplo:

- **Contenedor y contexto:** permite la configuración de los componentes de la aplicación y de la administración del ciclo de vida de un objeto Java, permitiendo que éste sea compartido por todos los componentes de la aplicación, que se cree una nueva instancia si la lógica de negocio lo requiere, etc.

- **Acceso a datos**: Spring cuenta con muchos módulos y drivers de acceso a base de datos, también llamados como Spring data. Así, existe un Spring Data para MongoDB, un segundo para MySQL, etc. La práctica totalidad de las bases de datos más utilizadas tienen su particular conector Spring.

- **Gestión de transacciones:** unifica distintas APIs de gestión y coordina las transacciones para los objetos Java, permitiendo definir políticas de retorno en caso de error para, por ejemplo, volver al estado anterior tras una transacción fallida en una base de datos de naturaleza bancaria.

- **Modelo Vista Contenedor (MVC):** implementación de este paradigma o patrón de diseño del desarrollo de software, definiendo diferentes modelos o entidades asociadas a un Controlador o Servlet.

  A través de este controlador se definen diferentes llamadas REST que permiten modificar los valores del Modelo, u obtener sus atributos, para luego ser *renderizados* en una Vista que será expuesta a posteriori en el Portal Web que la emplee.

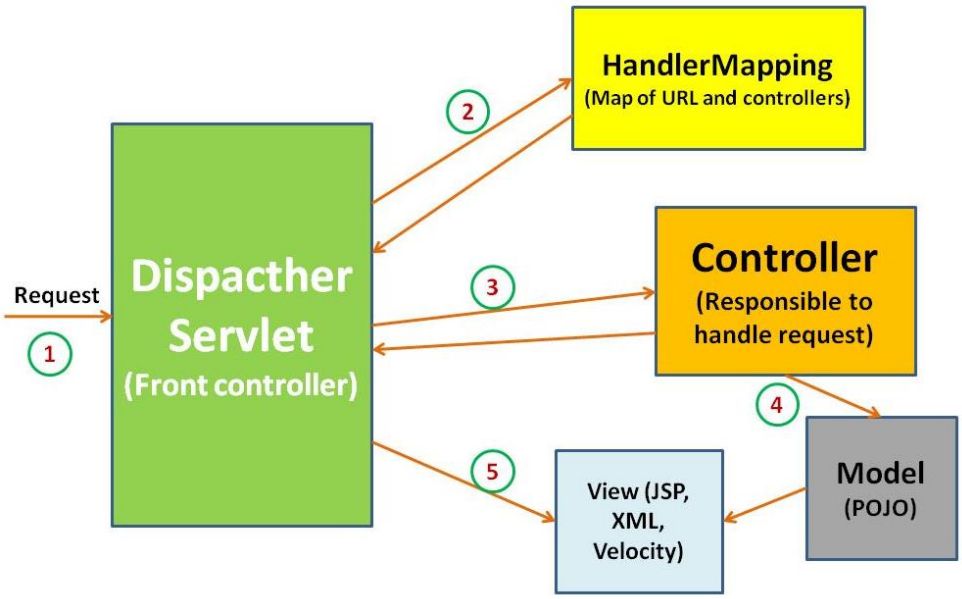

Figura 7: arquitectura Spring MVC

El módulo *Dispatcher Servlet*, que realizará un primer chequeo de formato de la petición recibida para descartarla en primera instancia si su formato, parámetro, cabeceras HTTP… son incorrectos. *Dispatcher Servlet* será el encargado de disparar la petición hacia el módulo *Handler Mapping*.

El módulo *Handler Mapping* analiza con más detalle la petición HTTP recibida, recogiendo la URI de la petición HTTP que nos llega y comprobando si está contemplada en el set de URIs que conforman los Servlets de la aplicación Web desplegada y construida con Spring MVC.

El módulo de *Controllers* procesa la petición HTTP sirviendo de punto de entrada a la lógica de negocio de la aplicación Web para, finalmente, devolver una respuesta.

La respuesta generada por Spring MVC va acompañada de un *Modelo* que, en este caso, será un objeto Java plano (POJO) que será *renderizado* en una *Vista*. La *Vista* puede adoptar formato HTML, JSON, XML, etc.

- ***Frameworks* de acceso remoto:** soporta diferentes protocolos como RMI [42] o JMX [43] que le permiten el acceso externo a la lógica de negocio, o a los objetos Java. Es una alternativa a los diferentes controladores y accesos por REST API.

- **Autenticación y autorización**: pone a disposición módulos de autenticación básicas que integran conexiones con base de datos que incluyen establecimientos de sesiones a través de *cookies*. Este módulo, llamado Spring *security*, viene complementado por un segundo módulo, Spring social, que integra autenticaciones con las redes sociales más populares del momento: Facebook y Twitter.

- ***Testing*:** incluye liberarías de testeo y prueba de errores del código desarrollado.

- **Soporte de diferentes formatos:** los controladores pueden retornar las consultas de sus REST APIs en diferentes formatos. Spring soporta los más empleados, entre ellos XML, JSON, HTML, etc.

3. Descripción del sistema

3.1 Arquitectura

UniAffinity cuenta con una arquitectura modular altamente escalable en la que todos los módulos mantienen comunicación entre sí fundamentalmente de dos maneras bien distintas:

- A través de APIs que emplean los protocolos HTTP/HTTPs. A través de HTTP los clientes podrán comunicarse con *UniAffinity*. A su vez, *UniAffinity* se conectará con el exterior – Facebook, Twitter – empleado el mismo protocolo.

- A través de conectores o drivers que emplean el modelo TCP/IP [44] para mantener establecer y mantener conexiones con base de datos. En nuestro caso, MongoDB.

Dos de los módulos fundamentales de la arquitectura *UniAffinity*, *UniAfinity Solr server* y *UniAffinity Backend* - que a continuación trataremos -, han de instalarse bajo el contexto de un Servidor de Aplicaciones.

Existen un buen número de servidores de aplicaciones *opensource* que pueden llevar a cabo esta tarea con efectividad, como *Jetty* o *Apache Tomcat*, y cualquiera de las dos alternativas son buenas.

Para la arquitectura de *UniAffinity* se ha elegido *Apache Tomcat* como contenedor y servidor de aplicaciones, debido a:

- Veteranía y fiabilidad del proyecto *opensource*.

- Buena documentación.

- Instalación y configuración sencillas.

El diseño de la arquitectura del sistema debe responder a las siguientes necesidades que cada vez son más básicas y fundamentales hoy día en el desarrollo de software en lado servidor:

- **Modularidad:** deben extraerse desde el punto de vista técnico y desde el punto de vista de la lógica de negocio del proyecto el mayor número de módulos, o cajas negras, independientes. El sistema adoptará una configuración modular 1…N donde existirá un mínimo de un módulo de cada tipo para que el sistema funcione correctamente y a pleno rendimiento.

- **Escalabilidad**: un diseño modular permitirá, en caso de un gran número de usuarios registrados, o un aluvión de peticiones a *UniAffinity API*, un despliegue en horizontal de la arquitectura del sistema, aumentando el número de módulos de *Backend* que se estimen oportunos para dar solución inmediata a la sobrecarga del sistema.

- **Comunicación y balanceo de carga:** la comunicación entre los diferentes módulos deberá soportar balanceo de carga. Es decir, las conexiones entre los diferentes módulos permitirán una comunicación punto a punto flexible, donde el destino pueda ser un módulo cualquiera de los 1…N que formen parte del sistema.

Así, y para ofrecer una información algo más visual del diseño que se propone, a continuación podremos observar qué módulos forman parte del diseño del sistema al completo, visto desde una perspectiva general sin entrar en detalle:

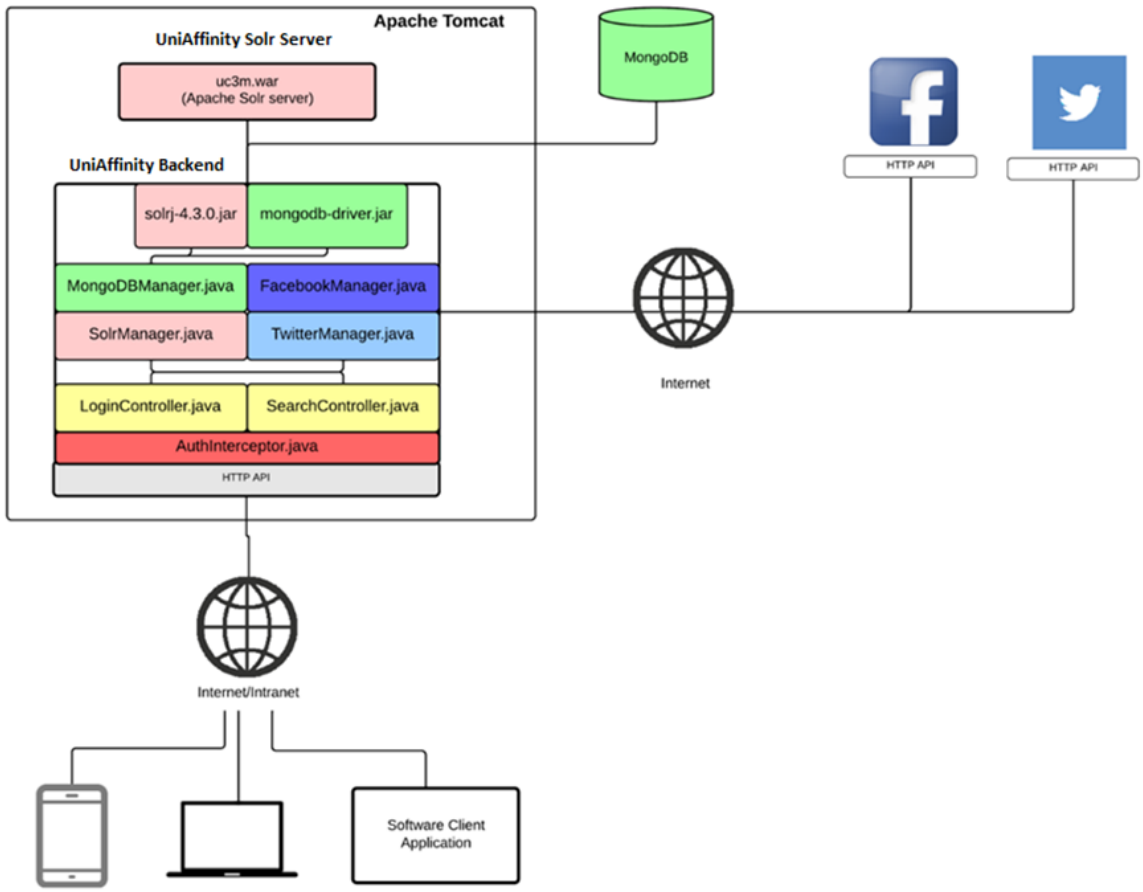

Figura 8: Arquitectura UniAffinity

En ella podemos encontrar los siguientes agentes:

- **UniAffinity Backend:** unidad lógica del sistema que ejerce de punto de conexión entre el Cliente y los diferentes recursos involucrados en la arquitectura, como MongoDB o Solr.

  UniAffinity Backend está desarrollado enteramente en Java bajo una arquitectura basada en Spring MVC para la definición de servicios y capas de Software y Maven [45] como gestor de dependencias.

  Implementa *UniAffinity API* como capa HTTP REST de comunicación externa, e internamente integra incluye conectores con MongoDB y Solr que, junto con la lógica de negocio y toda la casuística relacionada con *UniAffinity*, permiten registrar usuarios a través de Twitter y Facebook y acceder y relacionar sus datos de una manera sencilla, cómoda y rápida.

- **UniAffinity Solr Server:** servidor Apache Solr personalizado para almacenar los datos de los usuarios que pertenezcan a *UniAffiity*. *UniAffinity Solr Server* recibirá peticiones de búsqueda de *UniAffinity Backend* para la resolución de recomendaciones de usuarios, búsqueda de usuarios, etc.

- **Servidor de aplicaciones *Apache Tomcat***: contenedor de despliegue de aplicaciones Web. Incluirá en él los módulos *UniAffinity Backend* y *UniAffinity Solr Server*. Ninguno de estos dos módulos tienen por qué ser instalados en el mismo servidor Apache Tomcat, por lo que podrían desplegarse en máquinas distintas con servidores Apache Tomcat distintos.

- **Base de datos MongoDB**: módulo de almacenamiento persistente de datos en el que guardaremos los datos personales de nuestros usuarios extraídos de Facebook y Twitter. El acceso a base de datos será a través de su driver de conexión nativo a base de datos, *mongodb-driver.jar*.

  La instalación de MongoDB puede llevarse a cabo en una máquina distinta (o máquinas distintas, en caso de hacerlo en configuración *replica-set* o *cluster*) a la del servidor de aplicaciones Apache Tomcat.

  Este hecho refleja el enfoque modular y escalable de la arquitectura de UniAffinity, pensada para una instalación completa en una sola máquina servidora, o distribuida en diferentes máquinas servidoras.

- **Aplicaciones cliente**: diferentes aplicaciones que podrían hacer uso de *UniAffinity API*. Se citan algunos ejemplos como un Portal Web, una aplicación móvil y otros módulos de *Backend* que para su lógica de negocio vean interesante o necesario interactuar con *UniAffinity API*.

3.2 Requisitos funcionales

UniAffinity API tiene como propósito servir como pasarela de comunicación con las redes sociales Facebook y Twitter ejerciendo como módulo de autenticación por OAuth que, además, obtiene datos personales de los usuarios y ofrece búsquedas por afinidad entre usuarios estableciendo lazos de parentesco entre los datos obtenidos de unos y otros usuarios.

Su objetivo final subyace en ofrecer un interfaz de comunicación común y universal a diferentes aplicaciones de *software* clientes sea cual sea el lenguaje de programación que hayan empleado para su diseño y programación.

En la parcela de relación social entre usuarios su misión primordial es, pues, la búsqueda de usuarios afines, pero los datos que permiten esos resultados son de alta utilidad y por ello la fase de análisis anima a ofrecer otras vías de consumo alternativas para software cliente a través de búsquedas de texto libre sobre la base de conocimiento de los usuarios.

Como cualquier otro API, *UniAffinity API* debe ser cerrada en el sentido de que las operaciones que permita ejecutar han de estar fijadas. Sin embargo, debe ser un API flexible, abierto a las necesidades del cliente, ofreciendo diferentes tipos de operaciones o peticiones que recojan no sólo resultados de afinidad para un sujeto dado, sino también algunas otras operaciones que permitan explotar libremente los datos de los usuarios que hayan sido almacenados, como por ejemplo la consulta de usuarios afines (*affinity*) o bien operaciones de búsqueda de texto libre sobre campos específicos de usuarios (grupos de Facebook, tweets, etc.).

En el capítulo *4. Desarrollo* se detallarán todas las operaciones implementadas en *UniAffinity*, quedando mucho más claras las oportunidades que ofrece y el potencial de su API.

3.3 Requisitos no funcionales

Además de la comunicación entre el API y las aplicaciones cliente que consuman de ella, *UniAffinity* debe tener acceso a las diferentes APIs de las redes sociales de las que va a consumir: Facebook y Twitter.

Es por ello de obligado cumplimiento que el software de *UniAffinity* corra sobre algún tipo de plataforma que le permita acceso externo a Internet a través del protocolo HTTP, pues de lo contrario el proceso de Minería de Datos a partir del consumo externo de estos datos no podría llevarse a cabo.

Las peticiones externas que consuman del API de Twitter y Facebook tienen dos restricciones fundamentales muy a tener en cuenta.

- **Autenticación por OAuth:** tratado en puntos anteriores de la memoria técnica de este proyecto, sabemos debemos contar con los *tokens* de autenticación adecuados para ambos APIs.

- **Restricción en el número de peticiones a APIs**: tanto Twitter como Facebook permiten que los desarrolladores hagan uso de sus APIs. Sin embargo, para evitar un abuso en el consumo tienen fijados unos límites de peticiones máximos asociados por IP del cliente y por el token del mismo. Las limitaciones son:

    - Twitter REST API: 350 peticiones cada 15 minutos.

    - Facebook Graph API: no existen datos oficiales revelados por la propia red social, pero entre la comunidad de desarrolladores se calcula que el número máximo de peticiones es de entorno a las 60.000/día por *token* e IP.

3.4 Diseño

3.4.1 Persistencia en base de datos

UniAffinity emplea MongoDB como base de datos no relacional (*NoSQL*) para el almacenado de datos de usuarios de un modo persistente. Todos los datos que se hayan podido recuperar serán almacenados en MongoDB por los siguientes motivos:

- **Esquema mutable:** Los datos provenientes de Twitter o Facebook no guardan relación entre sí. Además, los datos en sí mismos son cambiantes desde el punto de vista del usuario, pudiendo un usuario tener un perfil mucho más completo que un segundo usuario por contar el primero con más actividad en las redes sociales.

  Además, los APIs de Twitter o Facebook son también cambiantes, pudiendo modificar el número de parámetros, naturaleza del contenido de los mismos… sin aviso. Es por ello que MongoDB, debido a su naturaleza sin esquema, es una buena alternativa para el guardado de datos *desnormalizado*, no relacional, y con un esquema mutable.

- **No relacional**: el usuario es la única entidad existente en nuestro sistema, y no existen como tal relaciones entre usuarios fijas. Todas las relaciones de similitud entre usuarios se calculan en tiempo de ejecución por cada petición que se haga a *UniAffinity API*, pues el incremento de nuevos usuarios registrados señalan que este tipo de relaciones de parentesco se lleven a cabo en tiempo real. Dado que no existen más entidades de datos en nuestro sistema, y que la única relación en el modelado de datos se lleva a

cabo en tiempo de ejecución, MongoDB como base de datos no relacional continúa siendo una buena alternativa.

- **Registros con tiempo de expiración**: como veremos más adelante, sería útil que nuestra base de datos tuviera la facultad de almacenar datos con un determinado tiempo de vida, también llamado *TTL*, que se eliminen cuando el tiempo de vida expire. MongoDB cuenta con esa funcionalidad, reforzando su candidatura.

- **Escalable:** desconocemos si *UniAffinity* tendrá éxito o no. Si el sistema alcanza un número alto de usuarios registrados y el tamaño de los datos almacenados crece cada día, nuestra base de datos debería soportar diferentes tipos de configuración que permitan escalar horizontalmente nuestro sistema, asegurándonos que disponemos siempre de al menos una instancia de la base de datos activa y accesible. MongoDB soporta configuraciones maestro-esclavos y en *cluster*.

3.4.2 Análisis de contenido textual

UniAffinity utiliza Apache Solr como motor de búsqueda para extraer resultados en base a una búsqueda realizada y como base para el algoritmo de similitud entre usuarios. Tanto la búsqueda de resultados dada una búsqueda de texto libre, o la relación entre usuarios a partir de su contenido volcado en Facebook y Twitter, se lleva a cabo a través de un análisis del contenido de texto.

Este análisis se fundamenta en el descubrimiento de términos dentro de un conjunto de texto dado. Si bien podríamos ir un paso más allá y no sólo descubrir términos, sino también etiquetar qué representa cada uno de estos términos (sustantivo, verbo, adjetivo, etc.) en, por ejemplo, un tweet de un usuario, lo cierto es que una separación de términos – también llamada *tokenización* – es suficiente para nuestra meta.

La separación o *tokenización* en términos más clásica implica la identificación de espacios en blanco en un texto para distinguir los términos a partir de él. Asimismo, se pueden aplicar técnicas más complejas para mejorar o enriquecer este proceso:

- *Tokenización* por expresiones regulares.

- Pre-filtrado de determinados caracteres para ser reemplazados por otros, o por vacío.

- Post-filtrado a partir de un listado de palabras (*stopwords*): este post-filtrado nos permitiría, por ejemplo, eliminar un listado de preposiciones en español.

- Post-filtrado para identificación de sinónimos a partir de una lista de sinónimos: este post-filtrado nos permitiría identificar un término y cambiarlo por un sinónimo bien

conocido por nosotros que figure en nuestra lista de sinónimos para ese término de origen.

En concreto, *UniAffinity* empleará un procesado de términos que aplicará las siguientes reglas:

1. Sustitución de algunos caracteres (comillas de todo tipo, signos de admiración e interrogación, etc.) por espacio en blanco.

2. *Tokenización* a través de una expresión regular que incluye: espacios en blanco, puntos, comas, punto y coma, arroba, guiones, etc.

3. Transformación de todos los términos a caracteres en minúscula.

4. Eliminación de términos duplicados.

Cada documento que indexemos en Apache Solr aplicará este procedimiento para aquellos campos que a través de los cuales se vaya a buscar. En ocasiones tendrá sentido no aplicar cadenas de procesos de este estilo para extraer términos puesto que quizá únicamente queramos tener determinados campos en nuestros documentos indexados que sirvan de presentación, no de búsqueda, como podría ser quizá un campo que represente número de teléfono o una cuenta de correo.

Para terminar de comprender cómo una expresión de búsqueda encaja con los diferentes documentos vamos a indexar en Apache Solr, acudamos al siguiente ejemplo clarificador:

1. Recoger la expresión de búsqueda de texto libre. Ejemplo:

```
Restaurantes en Madrid
```

2. Separar la expresión de búsqueda en términos aplicando los procedimientos descritos anteriormente. Ejemplo:

```
[0] restaurantes

[1] en

[2] Madrid
```

3. En el índice de Solr los documentos indexados mantendrán el contenido separado en términos, a su vez. Ejemplo:

Documento de ejemplo:

Campo identificador: 1234

Campo título: Restaurante La Toscana
Campo descripción: ¡Bonito restaurante italiano en Madrid!

Términos localizados por cada campo del documento:

Identificador: 1234
Título: restaurante, la, toscana
Descripción: bonito, restaurante, italiano, en, madrid.

4. Buscar los términos en el índice, donde la relación es de uno a uno. Es decir, tendremos que ver qué términos de nuestra expresión de búsqueda encajan con los términos de los documentos que tenemos indexados. En este caso, los términos de búsqueda `restaurante`, `en` y `madrid` se encuentran en el campo `descripción` de uno de los documentos del índice, por lo que ese documento sería un resultado de búsqueda positivo para dicha consulta.

Este análisis de texto en tiempo de consulta y en tiempo de indexación nos permitirá establecer relaciones de contenido y obtener resultados de búsqueda acordes con lo que esperamos encontrar, siempre que exista resultado alguno.

En función de nuestros requisitos, podemos ser más restrictivos a la hora de ejecutar consultas. Así, podríamos obligar a que los resultados de búsqueda tuvieran que tener los tres términos en su documento, o podríamos ser más flexibles y optar por que uno, dos o los tres se encuentren en el documento. Al primer comportamiento se le asemeja con una puerta lógica *AND* (este término, y este término, y este…), mientras que el segundo aplica a una puerta lógica *OR* (este término, o este término, o este…).

3.5 Casos de uso

Todos los casos de uso de *UniAffinity* contemplan dos instalaciones de *UniAffinity* distintas preparadas para su explotación en un entorno de producción real:

1. **Servicio SaaS en la nube:** instalación en uno o varios servidores remotos para que cualquier cliente pudiera hacer uso de *UniAffinity*. Tal y como se avanzará en el capítulo de Desarrollos futuros, esta alternativa requeriría un desarrollo menor que incluiría una clave identificadora por cada cliente para que los clientes sólo puedan insertar y consumir datos de su aplicación, no de otras.

   Este identificador sería único por cada cliente y serviría para identificar cada sistema cliente que consuma de *UniAffinity* en modo SaaS. Además, serviría también para

controlar el número de peticiones al API, algo útil si el modelo de negocio gira en torno al pago por número de peticiones diarias o mensuales.

2. **Instalación en modo *appliance*:** instalación en los sistemas de la aplicación cliente para que el uso y consumo sea único y exclusivo del cliente

3.5.1 Portal Web

El caso de uso más directo sería el desarrollo o la integración de *UniAffinity* en un portal Web. La integración en portales ya existentes asumiría que éstos incluyen algún tipo de gestión de usuarios, o que tras la integración de *UniAffinity* en su sistema desean tenerla.

Si el enfoque parte de un desarrollo de un portal Web desde cero, el portal podría adoptar un sinfín de temáticas bien diferenciadas donde la gestión de usuarios sea una funcionalidad a tener en cuenta. Teniendo en cuenta las funcionalidades y el valor añadido que *UniAffinity* podría aportar, se sugieren las siguientes temáticas:

1. **Portal Web de estudiantes universitarios de la Comunidad de Madrid:** temática estudiantil con enfoque de ocio en el que los estudiantes puedan conocerse entre sí y establecer amistad, compartir apuntes, buscar compañeros de prácticas, etc.

2. **Portal Web deportivo:** temática de comunidad deportiva con componente regional en la que los usuarios puedan buscar y encontrar compañeros para la práctica deportiva, establecer lazos de amistad, promover competiciones deportivas, etc.

3. **Portal Web E-commerce:** tienda *online* que, por definición, incluya una gestión de usuarios que puedan realizar transacciones en el portal. UniAffinity podría aportar al propietario del E-commerce ese sistema de registro y autenticación de usuarios junto con un valor añadido muy interesante: conocer con más detalle los gustos y aficiones de su clientela, pudiendo adaptar su stock y sus ofertas al perfil de los usuarios.

4. **Portal Web de *Dating* (Citas):** comunidad de usuarios en la que éstos podrán buscar y encontrar amigos, compañeros y, por qué no, pareja.

Tanto la integración de *UniAffinity* en un portal Web ya existente como el desarrollo de uno nuevo, las funcionalidades extra con las que contaría el portal serían:

1. Registrar usuarios a través de Twitter y Facebook con UniAffinity como pasarela.
2. Buscar usuarios similares a uno mismo.

3. Búsqueda de texto libre que aplicara a la información contenida en tweets, descripciones, grupos y *likes* de los usuarios.

4. Mostrar resultados de búsqueda incluyendo datos personales útiles para la creación de vistas previas y vistas detalladas de los perfiles de usuarios de los resultados de búsqueda, incluyendo:

    a. Fotografía.
    b. Nombre.
    c. País.
    d. Localidad (opcional).
    e. Sexo (opcional).
    f. Edad (opcional).
    g. Descripción (opcional).

En este caso de uso tanto la instalación de *UniAffinity* en SaaS como en modo *appliance* serían compatibles.

3.5.2 Aplicación móvil

De modo similar al portal Web, se plantea el caso de uso de integración de *UniAffinity* en alguna aplicación móvil ya existente - Android como iOS o Windows Phone -, incluyendo todas las posibilidades y funcionalidades que *UniAffinity* aporta.

Un amplio porcentaje de aplicaciones móviles emplean autenticación directa con Facebook y Twitter para ofrecer un servicio completo de cara al usuario. Si la autenticación fuera a través de *UniAffinity* no sólo conseguirían ese propósito, sino que podrían disfrutar de toda la información relativa al usuario y, si encaja con la temática de la *App*, también de *UniAffinity* API como complemento social a su aplicación.

En este caso, de nuevo tanto la instalación en modo SaaS como en modo *appliance* serían compatibles. Sin embargo, dado el enorme número de usuarios que desarrollan aplicaciones para terminales móviles que no cuentan con infraestructura de servidores, tendría mucho más sentido ofrecer *UniAffinity* como servicio en la nube (SaaS).

3.6 Minería de datos

Los APIs de Twitter y Facebook ofrecen una gran cantidad de información para extraer. Del volumen total de los datos disponibles recogeremos, para cada uno de ellos, los siguientes valores de los datos de los usuarios que se consideran útiles, especialmente para la definición del algoritmo de similitud entre usuarios a partir de éstos.

Entrar a definir qué se puede extraer de cada uno de los APIs de ambas redes sociales nos llevaría cientos de páginas. Existe una documentación oficial que detalla con exactitud todas las operaciones, formatos y datos que se puede recuperar de cada una de ellas.

A continuación se citan aquellos parámetros que *UniAffinity API* emplea en su lógica de negocio para almacenar la información del usuario y para la definición posterior del algoritmo de similitud entre usuarios.

3.6.1 Facebook

- **`FacebookUserName`**: nombre del usuario del perfil de Facebook.
- **`FacebookBirthDay`**: fecha de nacimiento del usuario indicado en el perfil de Facebook.
- **`FacebookEmail`**: correo electrónico indicado por el usuario en el perfil de Facebook.
- **`FacebookImgUrl`**: URL que apunta al estático de la imagen empleada por el usuario en su perfil de Facebook.
- **`FacebookProfileUrl`**: URL que apunta al perfil de Facebook del usuario.
- **`FacebookName`**: nombre real indicado por el usuario en el perfil de Facebook.
- **`FacebookGender`**: sexo del usuario indicado en el perfil de Facebook.
- **`FacebookLocale`**: lenguaje por defecto configurado por el usuario en su perfil de Facebook.
- **`FacebookTimeZone`**: huso horario del usuario indicado en el perfil de Facebook.
- **`FacebookHometown`**: localidad de origen indicada por el usuario en el perfil de Facebook.
- **`FacebookLocation`**: localidad actual donde vive el usuario indicada en el perfil de Facebook.
- **`FacebookEducation`**: listado de centros educativos (colegios, institutos, universidades…) que figuran en el perfil del usuario en Facebook.
- **`FacebookLanguages`**: listado de idiomas que el usuario habla indicado en el perfil de Facebook.

- **`FacebookGroups`**: listado de grupos a los que sigue el usuario.
- **`FacebookLikes`**: listado de *likes* del usuario.

3.6.2 Twitter

- **`TwitterId`**: identificador numérico del usuario en Twitter.
- **`TwitterUsername`**: nombre del usuario en Twitter.
- **`TwitterDescription`**: descripción del usuario que figura en el perfil de Twitter.
- **`TwitterImgUrl`**: URL que apunta al estático de la imagen del perfil del usuario en Twitter.
- **`TwitterHashtags`**: listado de *hashtags* que figuran en los últimos 100 tweets que el usuario ha publicado en el momento en que hace *login* con *UniAffinity API*.
- **`TwitterHashtags`**: listado de *hashtags* de los últimos 100 tweets marcados como favoritos por el usuario considerando como punto de inicio el momento en que hace login con *UniAffinity API*.
- **`TwitterFavouriteTweets`**: últimos 100 tweets íntegros marcados como favoritos por el usuario.
- **`TwitterTweets`**: últimos 100 tweets íntegros.
- **`TwitterMentioned`**: últimos 100 tweets en los que el usuario haya sido mencionado.
- **`TwitterFollowers`**: listado de los identificadores de usuario que siguen al usuario.
- **`TwitterFollowed`**: listado de los identificadores de usuario a los que el usuario sigue

3.7 Algoritmo de similitud

El movimiento del análisis de grandes volúmenes de datos con fines de estudio de mercado, estrategias de publicidad (mercadotecnia), perfilado de usuarios para estudios demográficos,

campañas de publicidad orientada al usuario, etc., exigen la puesta en práctica de complejas estrategias de minería de datos y un posterior cribado y perfilado de, en nuestro caso práctico, usuarios.

Existen numerosas formas de establecer relaciones de parentesco entre dos entidades dentro de un mismo conjunto de datos. Muchas de las aproximaciones se implementan con complejos modelos estadísticos que, a través de procesos de Aprendizaje Máquina, permiten agrupar los datos que tengan cierta similitud dentro del total de la muestra del estudio.

En otras palabras, a través de un análisis estadístico existen técnicas que permiten estimar qué es similar a qué y, a su vez, reunir en grupos todas aquellas muestras – usuarios, en nuestro caso – que tengan parentesco entre sí.

La **primera aproximación** a dar solución a nuestra necesidad sería, pues, una **agrupación** o de datos a través de algoritmos de Aprendizaje Máquina, empleando la información del usuario como criterios de semejanza a partir de los cuales entrenar nuestro modelo de agrupación de usuarios.

Esta técnica, aunque interesante y de buenos resultados si el entrenamiento es el adecuado, cuenta con un gran inconveniente para *UniAffinity API*: para agrupar de manera automática individuos en base a su información debemos tener una base de usuarios razonablemente numerosa; con su información, entrenaríamos un modelo de agrupación de usuarios.

Este modelo, si la base de usuarios continúa creciendo, puede ser ineficiente, y debería volver a generarse una y otra vez para tener actualizada la base de conocimiento de los nuevos usuarios entrantes y así tener una recomendación acorde y enriquecida con la información adquirida de los nuevos usuarios registrados.

Estas tareas de entrenamiento suelen ser procesos exigentes y de duración elevada que, en definitiva, resultan poco flexibles en el software que aplica a *UniAffinity API*, pues exigiría una revisión y actualización constante del *Backend*. En otras palabras, si optáramos por esta vía de recomendación tendríamos que desplegar en el entorno de producción prácticamente a diario una actualización de nuestro software para mantener un sistema de recomendación actual y acorde con los nuevos usuarios que cada día se registren en *UniAffinity*. Esta práctica recurrente es ineficaz y pone en riesgo la estabilidad del sistema.

Existen herramientas de software libre como Apache Mahout [46] que permiten realizar estas tareas de manera distribuida sirviéndose de Apache Hadoop. Mahout cuenta con una librería de algoritmos de agrupación, clasificación y recomendación de usuarios desarrollada en lenguaje Java muy completa que podría servir a estos fines.

Asimismo, una **segunda aproximación** bien podría ser emplear **algoritmos de recomendación** a través de filtrado colaborativo que el mismo Apache Mahout dispone. En esencia, un algoritmo de recomendación colaborativa se resume con el siguiente ejemplo: la recomendación de productos que Amazon nos sugiere al revisar la ficha de algún producto en concreto. Internamente, Amazon estudia qué usuario (con un identificador asociado) suele ver o comprar

(cada producto contará, a su vez, con su identificador) para tener un conocimiento pleno de los hábitos de compra de sus usuarios a través de tuplas:

[Identificador de usuario | identificador de artículo]

De este modo, podemos desarrollar un método de recomendación que nos sugiera "*usuarios que han visto este producto también han visto…*". Podríamos llevar a cabo la recomendación de un modo inverso, e igualmente válido en función de nuestras necesidades, para *recomendar "productos que han sido vistos por este usuario también ha sido visto por estos otros usuarios"*.

Esta segunda recomendación podría encajar para dar solución a nuestro problema de relacionar usuarios con usuarios, pues podríamos admitir por razonable que si un usuario visita una serie de productos y un segundo usuario visita buena parte de los del anterior individuo, podríamos inducir que ambos individuos son similares por haber visitado productos iguales.

Pese a todo, los algoritmos de recomendación con filtrado colaborativo también emplean técnicas de Aprendizaje Máquina y, aunque son mucho más simples de desarrollar que uno por criterio de agrupación, exigen igualmente volver a entrenar el modelo que permite recomendar automáticamente. Estas tareas son de nuevo muy exigentes en recursos informáticos (*CPU, RAM*) y en tiempo. Se espera que *UniAffinity* pueda contar con un número de usuarios creciente, por lo que habría que volver a entrenar el modelo con frecuencia.

Ninguna de las dos propuestas anteriores satisface una de las necesidades más básicas de nuestro proyecto:

- Contar con un sistema de recomendación eficaz que esté siempre disponible y que no exista ninguna pérdida de servicio por labores de actualización en el sistema de recomendación.

- Contar con un sistema de recomendación que no tenga limitación alguna en el número de usuarios registrados para poder empezar a establecer recomendaciones. Idealmente debería poder empezar a recomendar usuarios entre sí si *UniAffinity* únicamente contara con dos usuarios con algún parentesco real.

3.7.1 Algoritmo UniAffinity

Nuestra aproximación se basará en el algoritmo *tf-idf* que nativamente Apache Solr utiliza para definir su algoritmo de relevancia para la ordenación de búsquedas de texto libre.

La relación entre usuarios tiene en cuenta en primer lugar los siguientes axiomas:

- Un usuario en *UniAffinity* tendrá su propio identificador único.

- Todas las peticiones hacia *UniAffinity API* incluirán una cookie que almacenará encriptado dicho identificador. A través de este identificador *UniAffinity Backend* podrá extraer toda la información almacenada para dicho usuario.

- Un usuario en Facebook contará con un identificador único y un nombre de usuario único. *UniAffinity* almacenará éste último, correspondiente el campo `FacebookUsername`. No se debe confundir este campo con el nombre de usuario en Facebook, pues ése sí podría ser usado por múltiples usuarios.

- Un usuario en Twitter contará con un identificador único (`TwitterId`) y un nombre de usuario único (`TwitterUsername`).

- Como ya sabemos, dentro de la información que almacenemos de un usuario de *UniAffinity* tendremos las referencias a los nombres de usuarios únicos de Twitter y de Facebook, si ha enlazado ambas cuentas; si no, al menos de una de ellas.

Teniendo en cuenta los puntos anteriores, pasamos a analizar la información recabada del usuario en ambas redes sociales. A pesar de que son muchos los campos de información rellenos que podemos extraer de un único usuario, debemos hacer un análisis que arroje luz sobre cuáles de todos ellos sirven de punto de referencia para comparar usuarios entre sí y buscar algún tipo de afinidad.

Por un lado, en Twitter consideramos que la actividad que un modo más básico define cuáles son las tendencias o aficiones de un usuario son:

- **Usuarios a los que sigue el usuario** (`TwitterFollowed`): listado de identificadores de usuario a los que el usuario sigue personalmente. Ésta es una declaración absoluta de intenciones por su parte y no debemos pasarla por alto.

- **Usuarios que siguen al usuario** (`TwitterFollowers`): usuarios que siguen a ése usuario. Del mismo modo, pero cambiando el foco, que determinada gente siga a uno mismo está definiendo qué tipo de persona somos, por qué sectores informativos nos movemos, qué solemos compartir, *twittear*, etc.

- **Hashtags en los tweets del usuario** (`TwitterHashtags`): los *hashtags* son etiquetas representativas que apuntan de un modo u otro a la temática o tendencia de un tweet. Definen en un solo término al tweet en sí mismo. Almacenar este tipo de información es vital para conocer con más detalle al usuario y sus tendencias de o temas escritura.

- **Hashtags de tweets marcados como favoritos** (`TwitterFavouriteHashtags`): que un usuario marque como favorito un determinado tweet es, de nuevo, una declaración de intenciones que le define a sí mismo. Así pues, y bajo el mismo argumento del punto

anterior, un *hashtag* de un tweet marcado como favorito representa un grado mayor de definición del usuario a través de, no ya de su tendencia escritora, sino acerca de qué le gusta leer y qué temas son de su interés.

Por otro lado, de toda la información extraída de Facebook elegiremos los siguientes campos a tener en cuenta para la elaborar el algoritmo *UniAffinity* de similitud de usuarios:

- **Likes del usuario** (`FacebookLikes`): quizá se trate, junto con el siguiente campo, el más obvio de todos ellos. Los likes de un usuario reflejan qué le gusta al usuario, y tenerlo en cuenta para relacionar usuarios que les haya podido gustar un mismo contenido es mandatorio.

- **Grupos del usuario** (`FacebookGroups`): nos encontramos ante el mismo razonamiento anterior.

- **Localización del usuario** (`FacebookLocation`): al contrario que con Twitter, podemos saber exactamente cuál es la ciudad de origen del usuario, no la ciudad actual. Establecer lazos de similitud teniendo en cuenta nuestro lugar de origen resulta razonable.

- **Lenguajes que habla el usuario** (`FacebookLanguages`): un usuario puede hablar uno o varios idiomas. Aunque de menor peso que los anteriores, establecer lazos de parentesco entre usuarios gracias a los idiomas que puedan hablar en común es un punto a tener en cuenta.

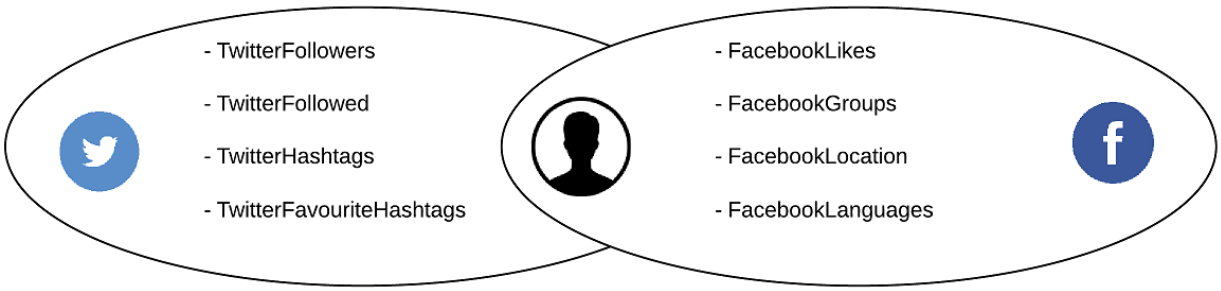

Figura 9: parámetros identificativos del usuario en UniAffinity

Con ambos orígenes de datos – Twitter, Facebook –, y habiendo cribado los campos considerando únicamente aquellos que definen el perfil, gustos, aficiones y los orígenes del usuario, agrupamos, por un lado, la información obtenida de Twitter, y por otro lado la extraída de Facebook. A estas dos agrupaciones las llamaremos en adelante súper-conjuntos. En concreto:

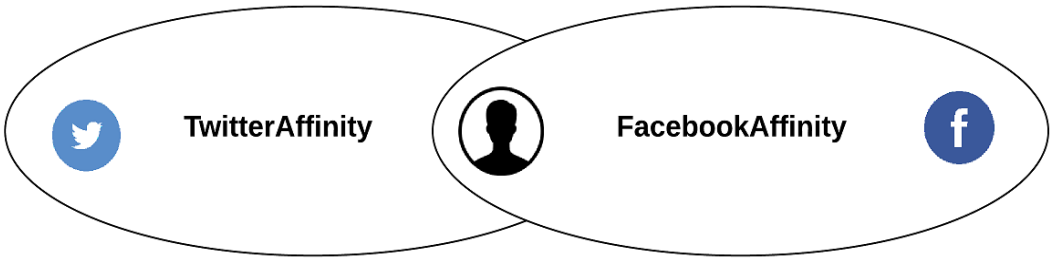

Figura 10: súper conjuntos de parámetros identificativos en UniAffinity

Cada uno de estos dos súper-conjuntos almacenará como valor el listado de términos que tengan por valor los parámetros que los componen. Es decir, `TwitterAffinity` tendrá como valor el contenido de `TwitterFollowers`, `TwitterFollowed`, `TwitterHashtags` y `TwitterFavouriteHashtags`. Análogamente, `FacebookAffinity` tendrá como valor una lista que incluirá el contenido de `FacebookLikes`, `FacebookGroups`, `FacebookLocation` y `FacebookLanguages`.

Como ya sabemos, Solr es un motor de búsqueda textual. En *UniAffinity Solr* Server cada uno de nuestros usuarios tendrá un documento indexado en Solr con toda la información del usuario, de modo que podremos ejecutar consultas de búsqueda sobre este índice de datos.

De este modo, y como ya vimos en el capítulo de *Diseño*, la búsqueda en Solr se hará uno a uno, analizando el texto de entrada, extrayendo sus términos, y buscando estos términos en el índice que compondrán cada uno de los documentos (usuarios) indexados.

El algoritmo de similitud de *UniAffinity* se fundamenta en el concepto de búsqueda en Solr, siendo los resultados de esta búsqueda – si los hay – los usuarios similares al de origen. Y es que, a fin de cuentas, un resultado de búsqueda no deja de ser una recomendación de resultados que podrían encajar con la expresión de búsqueda introducida.

La ejecución completa del algoritmo conlleva los siguientes pasos:

1. Como ya indicamos en los axiomas enumerados anteriormente, todas las peticiones a *UniAffinity API* deberán incluir una cookie identificativa (`uc3m`) que identificará al usuario consultante. Gracias a este identificador *UniAffinity Backend* construirá los súper-conjuntos `TwitterAffinity` y `FacebookAffinity` del usuario que busca usuarios similares a él.

2. *UniAffinity Backend* construye una consulta con la sintaxis de Apache Solr y la lanza contra el índice de usuarios indexados. En esta consulta buscaremos el contenido de los súper-conjuntos `TwitterAffinity` y `FacebookAffinity` del usuario original sobre los súper-conjuntos de los usuarios indexados. Como ya sabemos, el *matching* de términos será uno a uno.

3. La consulta ejecutada en Solr es muy rápida, y en apenas unos milisegundos habrá finalizado. Cualquier resultado positivo (documentos de Solr) indicará que hay usuarios similares consultante. Estos documentos serán ordenados por relevancia que, como sabemos, por defecto emplea el algoritmo *tf-idf* para el cálculo de la misma. Esto, en otras palabras, hará que el primer usuario que figure en la respuesta de Solr sea considerado el usuario más similar al usuario inicial, y así consecutivamente hasta el total de documentos (usuarios) que incluya la respuesta.

Esta solución permite contar con recomendaciones de usuarios prescindiendo de cualquier entrenamiento y re-entrenamiento de módulos de Aprendizaje Máquina (agrupación o recomendación), pues en el caso mejor necesitaremos únicamente el registro de dos usuarios en nuestro entorno para poder recomendarse entre sí como usuarios similares. Además, la potencia de Apache Solr hará que estas relaciones se lleven a cabo en apenas unos milisegundos incluso con un índice poblado por millones de usuarios registrados.

4. Desarrollo del proyecto

4.1 Autenticación y registro de usuarios

4.1.1 Gestión de cookies

UniAffinity API cuenta con una fuerte vocación al desarrollo Web, permitiendo ser consumida y utilizada por diferentes aplicaciones o portales Web que puedan ofrecer tanto el apartado visual como la gestión de usuarios que ofrece *UniAffinity*. El uso de *cookies* para la gestión tanto de la autenticación como de la sesión activa de usuarios en portales Web es una práctica ampliamente extendida que aporta cierto grado de flexibilidad:

- Debido a su popularidad, su manejo es bien conocido por la comunidad de desarrolladores.
- Existen multitud de herramientas y estrategias que permiten un crear, modificar y destruir *cookies* desde *Backend*.
- Permiten definir tiempos de vida de la propia cookie, algo útil para definir el tiempo de vida de la autenticación del usuario.
- La gestión es transparente al usuario y los diferentes navegadores ayudan en ella.

A su vez, el manejo de *cookies* también es discutido, principalmente debido a:

- Agujeros de seguridad.
- En aplicaciones móviles la gestión de la sesión de un usuario mediante *cookies*, si bien está soportada, no es una buena práctica.

Considerando los diferentes argumentos a favor y en contra, el manejo de sesiones de usuario por Cookies para acceder a un API se considera una opción aceptable, robusta y ampliamente extendida, por lo que *UniAffinity API* la adopta.

4.1.1.1 Algoritmo de cifrado

Almacenar valores de cierta relevancia en una *cookie* nunca debería hacerse en claro: un algoritmo de cifrado debería ofrecer cierto grado de seguridad y de recubrimiento de estos datos para que ningún usuario mal intencionado pudiera acceder a dicha información; ni tan siquiera, por seguridad, debería conocer qué se está almacenando en una cookie o su formato de datos, pues podría ser un agujero de seguridad explotable.

UniAffinity almacenará determinados datos asociados con la sesión del usuario y la caducidad de ésta. Para almacenarlo de una manera segura empleará una función de resumen *SHA256* junto con una *salt* que permita cifrar los datos a partir de ella. Esta *salt* será única en el sistema y con ella se cifrarán el contenido de todas las *cookies*.

El procedimiento para almacenar el valor de una de las *cookies* que emplearemos será el siguiente:

1. Obtendremos el identificador de usuario que el sistema haya asignado a dicho usuario durante o tras un proceso de autenticación. Recordemos que este identificador siempre será único. Para que sea único, generaremos un UUID *(Universally Unique Identifier)* que garantiza que sea aleatorio y único.

2. Codificamos en *Base64* el identificador de usuario.

3. Codificamos en *Base64* el tiempo máximo de vida asignado a una cookie por nuestro sistema que, por defecto, será de 14 días.

4. Generamos una clave (*key)*. Esta clave será un *hash* creado a partir de una función MAC con un algoritmo *SHA-256* que tendrá como clave nuestra *salt* y como valor de referencia para el *hash* la concatenación en *Base64* del identificador de usuario y del tiempo de expiración.

5. Generamos un *hash* definitivo de manera análoga al punto anterior, pero en esta ocasión emplearemos como clave el hash creado en el punto (4) en lugar de la *salt*.

6. Formateamos el valor final de la cookie, que contendrá los siguientes valores separados por el carácter "|":

Identificador de usuario (Base64)	*Tiempo de expiración (Base64)*	*Hash generado en el punto (6)*

Tabla 2: estructura de cookies

4.1.1.2 Cookie uc3m-anonymous

El diálogo de autenticación con las redes sociales incluyen a éstas como tercer agente involucrado en el proceso: por un lado contamos con el usuario, por otro lado contamos con el backend de *UniAffinity*, y por último los APIs de Twitter y Facebook con los que tanto *UniAffinity* como el usuario debe interactuar para completar un proceso de registro.

Como ya avanzamos previamente, y ampliaremos en detalle en el siguiente, desde el momento en el que el usuario inicia el diálogo de autenticación con *UniAffinity* y éste cede el testigo a Twitter o Facebook para solicitar el acceso de *UniAffinity* a sus datos, aparece un estado intermedio que podría concluir en registro, o no.

Este estado intermedio es representado por la *cookie* `uc3m-anonymous`, que almacena como contenido el formato y los valores indicados en el anterior punto. Cabe matizar que el identificador de usuario inicialmente es inexistente, porque el usuario aún no está identificado. El sistema de autenticación de *UniAffinity* crea un *UUID* aleatorio en el momento en que detecta que el usuario no está registrado, asociándolo junto al resto de valores a la cookie *uc3m-anonymous*.

Si el usuario cerrara el ciclo de registro ese identificador anónimo que *UniAffinity* ha generado terminará siendo el identificador del usuario registrado. Si, por el contrario, el usuario ya estuviera registrado previamente y su intención es la de autenticarse en el sistema, ese identificador aleatorio de *`uc3m-anonymous`* será ignorado y se recuperará el identificador original del usuario con una consulta a MongoDB.

4.1.1.3 Cookie uc3m

La cookie `uc3m` almacenará exactamente la misma información descrita para la cookie uc3m-anonymous. Esta cookie representará la sesión de un usuario autenticado y siempre tendrá que ser incluida en todas y cada una de las peticiones que se lancen contra *UniAffinity API*, dando derecho al usuario a recuperar la información del sistema:

- Búsqueda de usuarios.
- Recuperación de usuarios similares a uno mismo, que será indicado a través de la cookie.

Para una mejor comprensión del papel que las cookies cobran en el sistema *UniAffinity* se puede ver en Figura 11, describiendo de manera general el comportamiento del sistema.

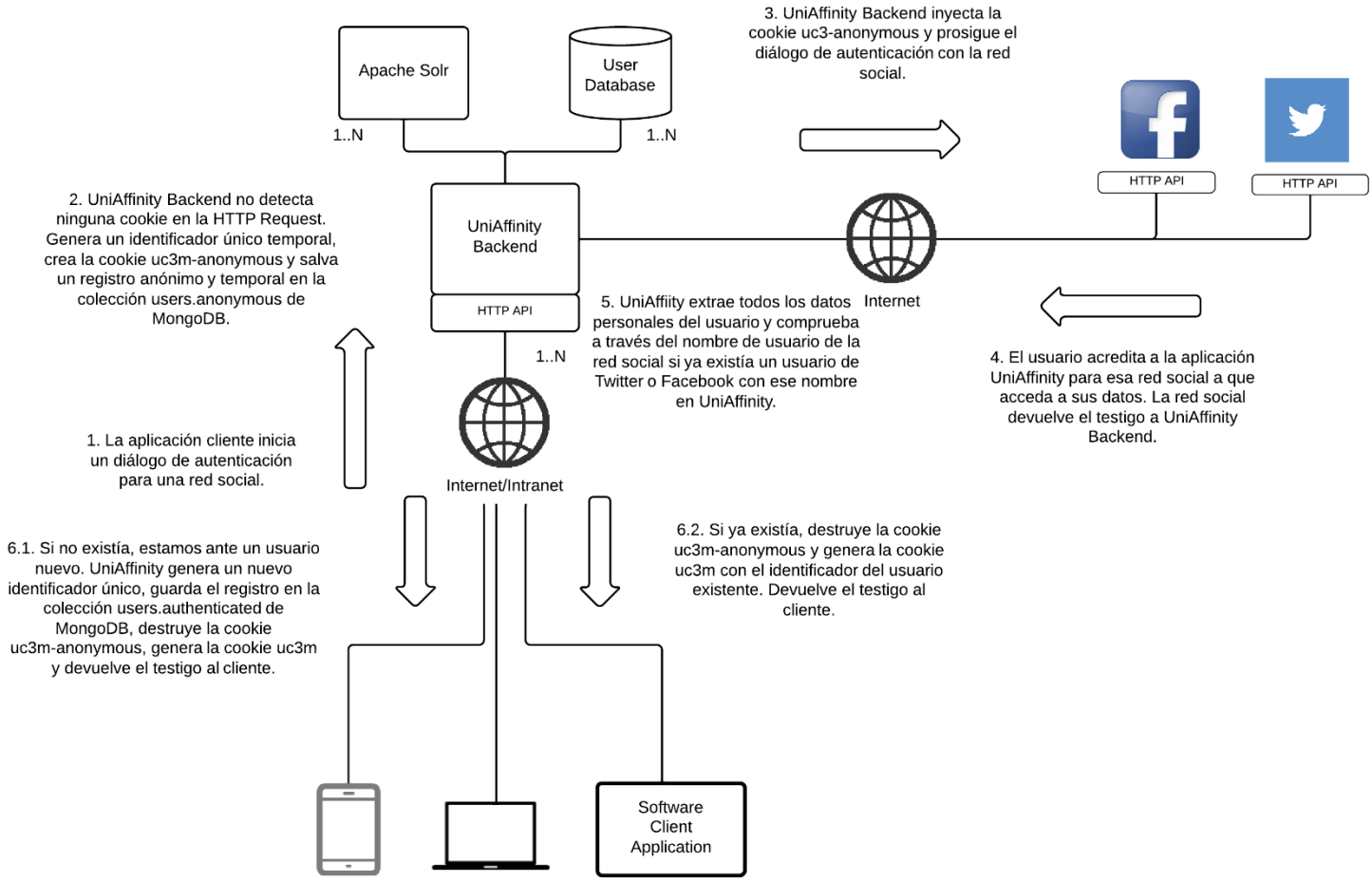

Figura 11: gestión de cookies durante el diálogo de autenticación

4.1.2 Registro de usuarios en el sistema

4.1.2.1 Registro con Twitter

1. *UniAffinity API* recibe una petición de autenticación con Twitter y llega a *UniAffinity Backend*.

2. Creamos un canal de comunicación desde k con Twitter indicando:
   a. `UniAffinity Twitter consumer key`: clave de autenticación de nuestra aplicación Twitter necesaria para interactuar con Twitter API. Esta clave es facilitada por Twitter.

b. *UniAffinity Twitter consumer secret*: clave de autenticación de nuestra aplicación Twitter necesaria para interactuar con Twitter API. Esta clave es facilitada por Twitter.

3. Realizamos una petición a Twitter desde *UniAffinity Backend* que, en base a los credenciales del punto (1), nos generará:

    a. Un *token*.

    b. Un *token secret*.

4. Comprobamos si la petición HTTP del Cliente incluye alguna cookie:

    a. **Si no incluye ninguna cookie** (*uc3m* o `uc3m-anonymous`) generamos un nuevo identificador aleatorio y la inyectamos en la respuesta HTTP.

    b. **Si contiene la cookie uc3m**, recogemos el identificador de usuario almacenado en el valor de la cookie y actualizamos en MongoDB los datos del usuario con el nuevo `token` y `token secret` asociados a él. Este paso es importante, pues ambos *tokens* tienen caducidad, y dado que el usuario ya estaba previamente autenticado cabe la posibilidad de que sus *tokens* estén ya caducados, por lo que actualizarlos es una buena práctica.

    c. **Si incluye la cookie uc3m-anonymous**, recuperamos el identificador de usuario y almacenamos en MongoDB una nueva entrada en la colección *users.anonymous*. Si la entrada ya existía previamente, se actualizará con un nuevo tiempo de vida (TTL). Si no existía, aparecerá como nueva entrada con el máximo TTL.

5. Acto seguido se redirige la petición a Twitter incluyendo en esta petición una URL de retorno o *callback* que apunta a *UniAffinity Backend*.

6. Twitter abrirá un formulario de validación en el que el usuario, si no estaba previamente autenticado en Twitter, tendrá que autenticarse en la red social. Posteriormente será preguntado si acepta que *UniAffinity* pueda acceder a sus datos personales.

    a. Si no acepta, Twitter redirige de nuevo a *UniAffinity Backend* con un código de error. *UniAffinity Backend* lo recoge, controla y propaga el error al Cliente indicando que no ha podido autenticar al usuario.

    b. Si acepta, Twitter redirige de nuevo a *UniAffinity Backend* con éxito. *UniAffinity Backend*, en este punto, tendrá que:

i. **Si la petición incluye la *cookie* uc3m**, el proceso de autenticación proviene de un usuario ya autenticado. Es decir, el usuario ya habrá sido autenticado previamente mediante este mismo proceso, o bien a través de Facebook.

   1. *UniAffinity* recoge el identificador de usuario que almacena la cookie uc3m y busca el usuario en MongoDB a partir de este identificador.

   2. Recibe al usuario y recoge su `token` y su `token secret`, previamente insertados en la primera parte del diálogo de comunicación.

   3. Abre una nueva vía de comunicación con Twitter indicando las claves de *UniAffinity* como aplicación de Twitter, así como también el `token` y token `secret` del usuario.

   4. Comienza a minar la información de usuario y la almacena en MongoDB.

   5. Si el almacenamiento en MongoDB es exitoso, indexa la información del usuario en Solr.

   6. Si la indexación es exitosa, inyecta una nueva *cookie* `uc3m` con valores de tiempo renovados para reactivar la sesión del usuario. Recordemos que la cookie tiene un tiempo de expiración asignado.

   7. Finalmente, devuelve una respuesta positiva al cliente en formato JSON indicando que la autenticación ha concluido exitosamente.

ii. **Si la petición incluye la cookie uc3m-anonymous**, extrae el identificador de usuario almacenado en ella y busca en MongoDB en la colección de usuarios anónimos y recoge los datos registrados previamente: identificador de usuario, `token`, `token secret` y fecha de inserción.

   1. *UniAffinity Backend* establece un nuevo diálogo de comunicación con Twitter indicando sus identificadores de aplicación de Twitter, así como el `token` y `token secret` del usuario anónimo.

   2. Obtenemos el nombre de usuario en Twitter asociado a su cuenta. El usuario que trata de autenticarse a priori es anónimo, pero no sabemos si previamente ha podido registrarse en *UniAffinity* y ha borrado las cookies del navegador, por lo que buscamos si hay previamente un usuario registrado con ese nombre de usuario en Twitter. Se lanza una consulta contra MongoDB para averiguarlo, y:

      a. Si el usuario existe, se recogen todos sus datos.

b. Si el usuario no existe, se crea un nuevo usuario con todos sus datos vacíos.

3. Extraemos toda la información del usuario en Twitter y se va rellenando toda la información.

4. El usuario será insertado en MongoDB.

5. Si el guardado es exitoso, se indexa la información de usuario en Solr.

6. Si el indexado es exitoso, se borra la cookie uc3m-anonymous y se genera una nueva *cookie* `uc3m`.

7. Se devuelve la respuesta al Cliente en formato JSON indicando que la autenticación ha sido exitosa.

Fin del proceso de autenticación con Twitter.

4.1.2.2 Registro con Facebook

1. *UniAffinity API* recibe una petición de autenticación con Facebook y llega a *UniAffinity Backend*.

2. Comprueba que la petición HTTP no incluya ninguna de las *cookies* del sistema.

   a. Si no contiene ninguna de ellas, genera un nuevo identificador aleatorio e inyecta en la respuesta HTTP una nueva *cookie* `uc3m-anonymous`. A diferencia del proceso de Twitter, no hay un estado anónimo persistido en MongoDB, pues no hay `token` ni `token secret` que almacenar a este nivel.

3. Redirige la petición HTTP hacia Facebook indicando el identificador de cliente de Facebook (aplicación de Facebook) y una URL de retorno o `callback` que apunte a *UniAffinity Backend*.

4. Facebook toma el testigo. Si el usuario no estaba previamente conectado a la red social, Facebook le pedirá que se autentique en ella. Acto seguido solicitará al usuario que dé permiso a *UniAffinity* a acceder a sus datos personales.

a. Si no acepta, Facebook redirige de nuevo a *UniAffinity Backend* con un código de error. *UniAffinity Backend* lo recoge, controla y propaga el error al Cliente indicando que no ha podido autenticar al usuario.

b. Si acepta, Facebook redirige de nuevo a *UniAffinity Backend* con éxito. *UniAffinity Backend*, en este punto, tendrá que:

   i. Recoge el `accessToken` y el parámetro *expires* de la respuesta que envía Facebook a *UniAffinity*.

   ii. **Si la petición incluye la cookie uc3m**, el proceso de autenticación proviene de un usuario ya autenticado. Es decir, el usuario ya habrá sido autenticado previamente mediante este mismo proceso, o bien a través de Twitter.

      1. Recoge el identificador de usuario de la *cookie* `uc3m`.

      2. Busca en MongoDB al usuario registrado con ese identificador.

      3. Actualiza los datos del usuario registrado minando su información de Facebook.

      4. Guarda el usuario con los datos actualizados en MongoDB.

      5. Si el guardado es exitoso, indexa en Solr los datos del usuario.

      6. Si el indexado es exitoso, regenera nuevamente la cookie uc3m actualizando el tiempo de expiración y envía una respuesta al Cliente en formato JSON indicando que la autenticación ha sido exitosa.

   iii. **Si la petición incluye la cookie uc3m-anonymous**, extrae el identificador de usuario almacenado en ella.

      1. *UniAffinity Backend* establece un nuevo diálogo de comunicación con Facebook indicando sus identificadores de aplicación de Facebook.

      2. Obtenemos el nombre de usuario en Facebook asociado a su cuenta. El usuario que trata de autenticarse a priori es anónimo, pero no sabemos si previamente ha podido registrarse en *UniAffinity* y ha borrado las *cookies* del navegador, por lo que buscamos si hay previamente un usuario registrado con ese nombre de usuario en Facebook. Se lanza una consulta contra MongoDB para averiguarlo, y:

         a. Si el usuario existe, se recogen todos sus datos.

b. Si el usuario no existe, se crea un nuevo usuario con todos sus datos vacíos.

3. Extraemos toda la información del usuario en Facebook y se va rellenando toda la información.

4. El usuario será insertado en MongoDB.

5. Si el guardado es exitoso, se indexa la información de usuario en Solr.

6. Si el indexado es exitoso, se borra la *cookie* `uc3m-anonymous` y se genera una nueva *cookie* `uc3m`.

7. Se devuelve la respuesta al Cliente en formato JSON indicando que la autenticación ha sido exitosa.

Fin del proceso de autenticación con Facebook.

4.1.2.3 Asociación de cuentas de Twitter y Facebook

Descritos los dos complejos flujos de autenticación de Twitter y Facebook, la asociación de cuentas de una y otra red social para un mismo usuario es una tarea trivial e implícita en el propio proceso de autenticación del usuario para una y otra red.

Si el usuario se ha autenticado y registrado con Twitter e inicia un proceso de autenticación con la Facebook, al tratarse de un usuario ya registrado e identificado por el sistema, su petición de autenticación hacia *UniAffinity API* contendrá la cookie `uc3m`, y *UniAffinity Backend* podrá extraer su identificador de usuario único en el sistema *UniAffinity*.

UniAffinity Backend continuará con el proceso de autenticado por Facebook como quedó reflejado en el punto anterior, con una única – pero vital – diferencia: el sistema sabe que el usuario está ya autenticado, pues cuenta con la *cookie* `uc3m` en su petición, y sabe quién es a partir de su identificador único.

De ese modo, tras extraer en su totalidad los datos personales de Facebook, éstos se emplean para actualizar los datos totales del usuario identificado, por lo que a la hora de salvar al usuario en MongoDB contendrá tanto los datos de Twitter como los de Facebook.

Es decir, si un usuario ya figura en el sistema y se autentica tantas veces como desee con Facebook o Twitter, sus datos no serán nunca eliminados, sino actualizados, y en *UniAffinity* ambas cuentas – junto con su información - siempre quedarán asociadas a él.

4.2 Backend

UniAffinity Backend es el módulo de negocio que, además, implementa la capa de comunicación de *UniAffinity API* a través de una HTTP REST API.

Se trata de un módulo desarrollado en Java empleando el *framework* Spring MVC, incluyendo clases o managers de comunicación con *UniAffinity Solr Server*, Facebook, Twitter y MongoDB.

El proyecto dispone de la siguiente estructura de paquetes:

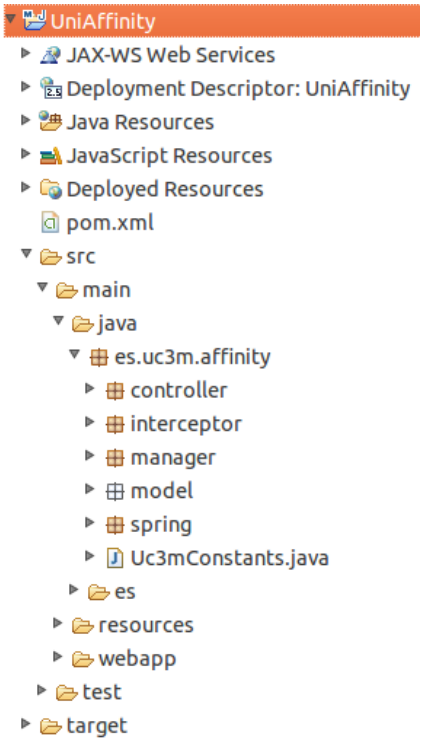

Figura 12: proyecto UniAffinity Backend

A nivel lógico encontramos las siguientes capas involucradas:

- **Interceptors**: contaremos con un único interceptor, `AuthInterceptor`, que filtrará las peticiones que incluyan la *cookie* `uc3m` y redireccionará a aquellas que no lo hagan o que contengan un formato incorrecto.

- **Controllers**: capa involucrada en la recepción de peticiones HTTP. Éste es el punto de partida del procesamiento de cualquier petición que haya sido validada por `AuthInterceptor`, el interceptor encargado de comprobar si se envía la *cookie* `uc3m` con el formado apropiado.

En esta capa encontraremos fundamentalmente dos *Controllers*: uno que se encargará de gestionar las peticiones de *UniAffinity API* y un segundo que tendrá como misión gestionar las peticiones de autenticación con las redes sociales.

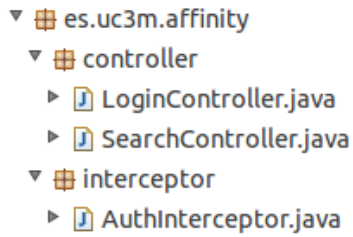

Figura 13: Controllers e Interceptor de UniAffinity Backend

- **Managers**: contendrán toda la lógica de negocio de nuestro *Backend*. En ellos se realizarán todas las operaciones necesarias para, por ejemplo, realizar todas las tareas de Minería de Datos de Twitter y Facebook o, la comunicación con MongoDB, o también las búsquedas contra el servidor Solr.

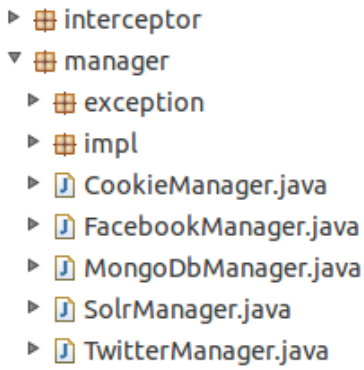

Figura 14: Managers de UniAffinity Backend

Si bien se trata de una capa lógica, *UniAffinity* también incluye el paquete:

- **Model**: incluye todas las clases Java que modelan una entidad o dato. A través de la sintaxis propia de Java, qué es un usuario, qué es un servidor Solr y, en definitiva, cualquier concepto necesario o útil para la lógica de negocio y el correcto funcionamiento del sistema.

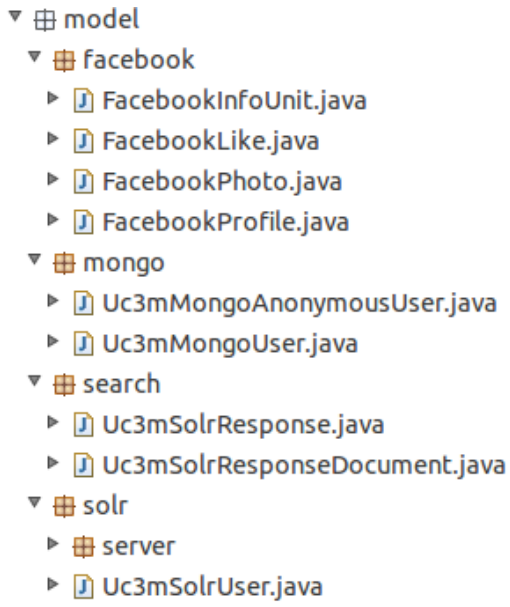

Figura 15: Modelos de UniAffinity Backend

En la clase `Uc3mConstants` se almacenan las claves de Twitter y Facebook que permiten establecer diálogo con sus respectivas HTTP APIs y las URLs de retorno o *callback* que apuntan a *UniAffinity Backend* que se ven involucradas en los diálogos de autenticación con ambas Redes Sociales.

Si bien podrían almacenarse en algún archivo de propiedades, o incluso en una base de datos para ser recuperadas en el momento del despliegue de la aplicación en un servidor de aplicaciones como Apache Tomcat, únicamente contiene seis propiedades. Debido al número bajo de propiedades a almacenar, y el contexto del proyecto *UniAffinity*, que se trata de un piloto en pruebas y no un software que vaya a ser desplegado en un entorno de producción accesible al público, se considera aceptable esta estrategia adoptada.

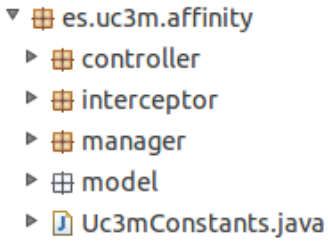

Figura 16: Uc3mConstants.java

Parte vital de *UniAffinity Backend* es el archivo `pom.xml` en el que se incluyen todas las dependencias de Maven que emplea el proyecto. Básicamente, Maven ofrece una serie de repositorios públicos en los que incluye las dependencias que la comunidad de Desarrolladores,

previo desarrollo, validación y pruebas de calidad, va completando con librerías de Software libre.

El archivo `pom.xml` de *UniAffinity Backend* incluye las librerías para el funcionamiento del proyecto, a destacar:

1. **Dependencias con Spring Framework**: todas aquellas dependencias (archivos `.jar`) que son necesarios incluir para dar, en definitiva, a *UniAffinity Backend* la estructura de aplicación Web que pueda ser desplegada posteriormente en Apache Tomcat.

2. **Driver de conexión con MongoDB**: conector oficial con MongoDB que permite comunicarse con la base de datos y ejecutar operaciones de lectura, escritura, actualización, etc.

3. **Driver de conexión con Solr (SolrJ)**: conector oficial con Solr que permite comunicarse con servidores Solr y ejecutar operaciones de búsqueda, indexación, etc.

4.2.1 Diagramas de clases

Para cada uno de los paquetes de clases Java anteriormente descritos se detallarán a continuación los diagramas UML de las clases que los componen. Si bien sería ideal contar con un diagrama conjunto de *UniAffinity Backend* que incluya todas las clases del proyecto, por claridad se ha preferido dividir los diagramas por paquetes.

4.2.1.1 Controladores (`Controllers`)

Figura 17: diagrama UML del paquete `Controllers`

En el paquete ***Controllers*** este encontramos las siguientes clases:

Clase	Descripción
`SearchController`	Implementa todos los endpoints de cada una de las operaciones de búsqueda de *UniAffinity* API.
`LoginController`	Implementa los *endpoints* de autenticación con Twitter y Facebook así como también los métodos expuestos a los *callbacks* tanto de Twitter como de Facebook que forman parte del diálogo de autenticación OAuth con ellos.

Tabla3: descripción de clases contenidas en el paquete `Controllers`

4.2.1.2 Manejadores (`Managers`)

En pro de una mejor visualización y entendimiento de los diagramas UML, la disposición de los diagramas y sus relaciones entre clases se ha fragmentado en dos partes, tal como se puede ver a continuación.

Managers: búsquedas y minería de datos en Twitter

En la Figura 18 se pueden ver un subconjunto de interfaces y clases del paquete `Manager` y las relaciones entre ellas. Más adelante en la Tabla 4 se describen las funcionalidades de dichas interfaces y en la Tabla 5 de las clases. Esta parte se encarga de la extracción de datos del usuario para la red Social Twitter y de la gestión de la indexación y búsqueda de documentos en nuestro servidor *UniAffinity Solr Server*.

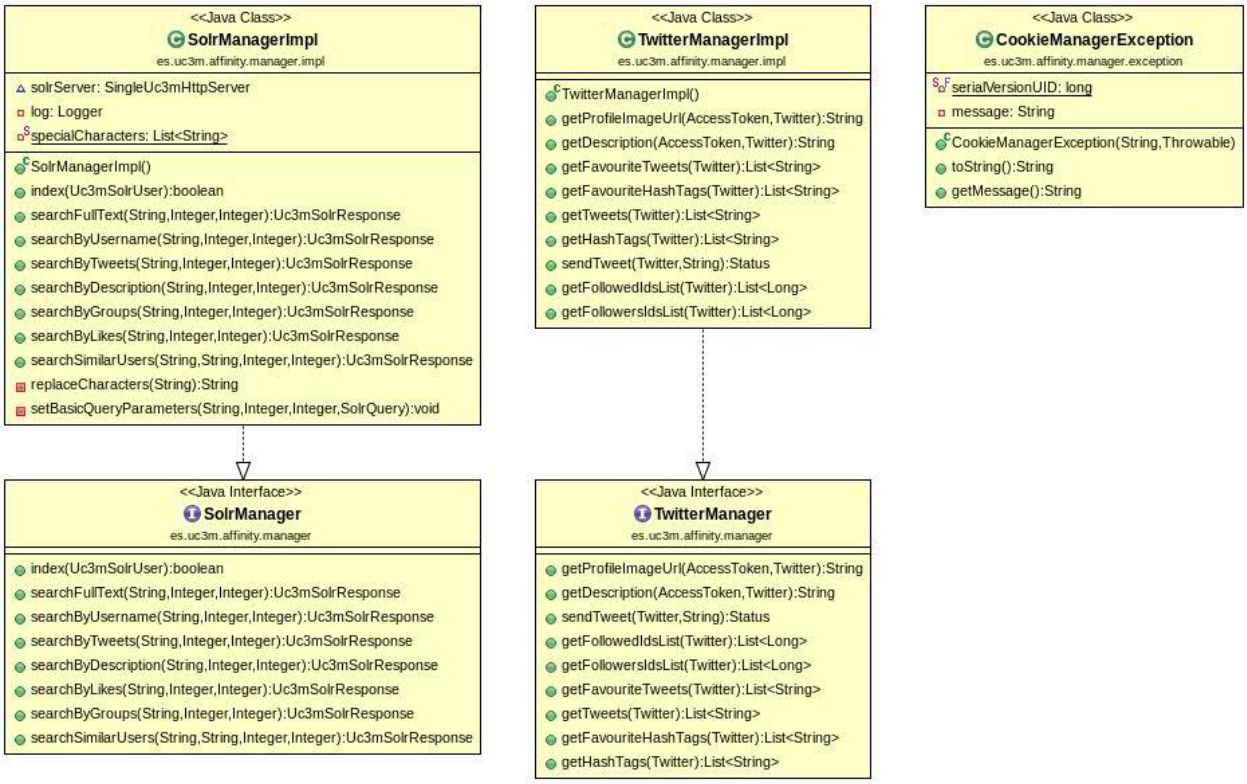

Figura 18: diagrama UML del paquete Managers (primera parte)

Los **interfaces** que se incluyen son:

Interfaz	Descripción
SolrManager	Define todas las operaciones de búsquedas que serán lanzadas contra el servidor *UniAffinity Solr Server*, junto con la operación de indexación de un nuevo documento en Solr que incluirá la información relativa al usuario.
TwitterManager	Define los métodos que permitirán extraer la información del usuario proveniente de la red social Twitter.

Tabla 4: descripción de interfaces contenidos en el paquete Managers (primera parte)

Las **clases** Java incluidas son las siguientes:

Clase	Descripción
SolrManagerImpl	Implementa el interfaz `SolrManager`. Desarrolla la lógica de cada uno de los métodos que permitirán indexar y realizar operaciones de búsqueda contra *UniAffinity Solr Server*.
TwitterManagerImpl	Implementa el interfaz `TwitterManager`. Desarrolla la lógica de cada uno de los métodos que permiten extraer información del usuario desde la red social Twitter.
CookieManagerException	Extiende la clase `Exception`. Define una excepción particular para el manager `CookieManager`.

Tabla 5: descripción de clases contenidas en el paquete Managers (primera parte)

Managers: búsquedas e indexación de información con Facebook

En la Figura 19 se pueden ver un segundo subconjunto de interfaces y clases del paquete `Manager` y las relaciones entre ellas. Más adelante en la Tabla 6 se describen las funcionalidades de dichas interfaces y en la Tabla 7 de las clases. Esta parte se encarga de la extracción de datos del usuario para la red Social Facebook, la gestión de cookies de *UniAffinity* y del almacenamiento de nuevos documentos en MongoDB.

Figura 19: diagrama UML del paquete Managers (segunda parte)

Los **interfaces** que se incluyen son:

Interfaz	Descripción
CookieManager	Define los métodos que permiten inyectar y borrar *cookies* `ucm3` y `uc3m-anonymous`,

FacebookManager	Define los métodos que permiten extraer la información personal del usuario de la red social Facebook.
MongoDbManager	Define los métodos que permitirán persistir los datos de un usuario tanto en la colección `users.anonymous` como los datos de usuarios registrados en la colección `users.authenticated`.

Tabla 6: descripción de interfaces contenidos en el paquete Managers (segunda parte)

Las **clases Java** incluidas son las siguientes:

Clase	Descripción
CookieManagerImpl	Implementa el interfaz `SolrManager`. Desarrolla la lógica de cada uno de los métodos que permitirán indexar y realizar operaciones de búsqueda contra *UniAffinity Solr Server*.
FacebookManagerImpl	Implementa el interfaz `FacebookManager`. Desarrolla la lógica de cada uno de los métodos que permiten extraer información del usuario desde la red social Facebook.
MongoDbManagerImpl	Implementa el interfaz `MongoDbManager`. Desarrolla la lógica de cada uno de los métodos que permite insertar y recuperar datos en MongoDB en las colecciones `users.anonymous` y `users.authenticated`.

Tabla 7: descripción de clases contenidas en el paquete Managers (segunda parte)

4.2.1.3 Modelos (`Models`)

Modelos: Facebook

En la Figura 20 se pueden apreciar un primer subconjunto de clases del paquete `models`. En este subconjunto se incluyen los diferentes conjuntos de modelos que describen la información del perfil del usuario de Facebook. Toda la información del perfil se ha generalizado en el modelo `FacebookProfile`. Más adelante, en la Tabla 8 se describen las funcionalidades detalladas de las clases.

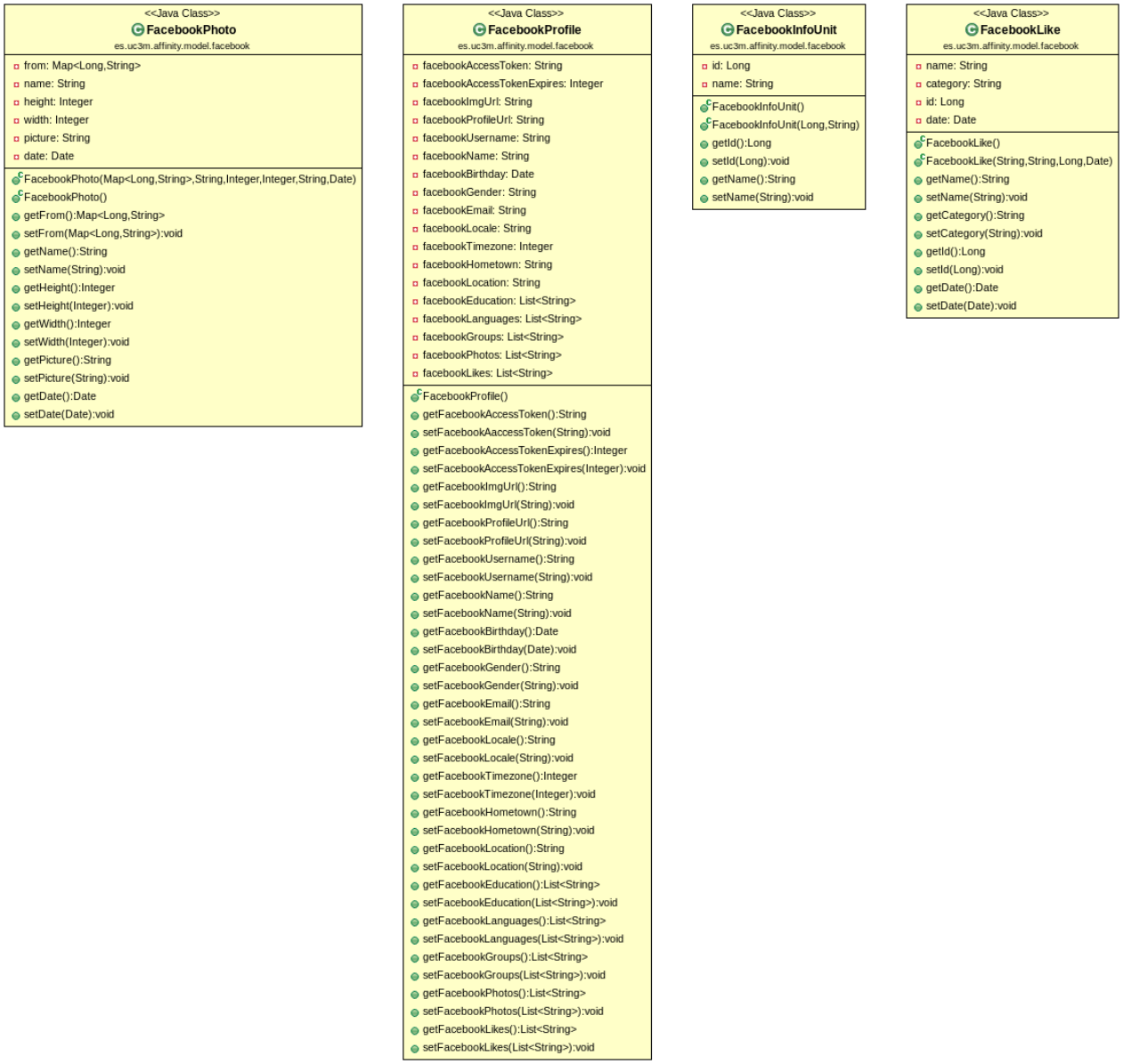

Figura 20: diagrama UML del paquete Models (primera parte)

Las **clases Java** incluidas son las siguientes:

Clase	Descripción
FacebookPhoto	Clase Java que define el modelo de información asociado a los datos que se pueden extraer desde Facebook relacionados con una imagen.
FacebookProfile	Clase Java que define el modelo de información asociado al perfil completo de un usuario de Facebook. Este modelo será el esqueleto que *UniAffinity* empleará para resumir la información total extraída de un usuario en Facebook.
FacebookInfoUnit	Clase Java que define el modelo de información asociado a los datos que se pueden extraer desde Facebook relacionados con un los datos informativos básicos de un usuario en Facebook.
FacebookLike	Clase Java que define el modelo de información asociado a los datos que se pueden extraer desde Facebook relacionados con un *Like* de un usuario en Facebook.

Tabla 8: descripción de clases contenidas en el paquete Models (primera parte)

En la Figura 21 se detalla el segundo subconjunto de clases que hacen referencia al modelado de datos del usuario relativo a MongoDB y a Solr. En la tabla posterior se detalla la funcionalidad de cada una de las clases implicadas.

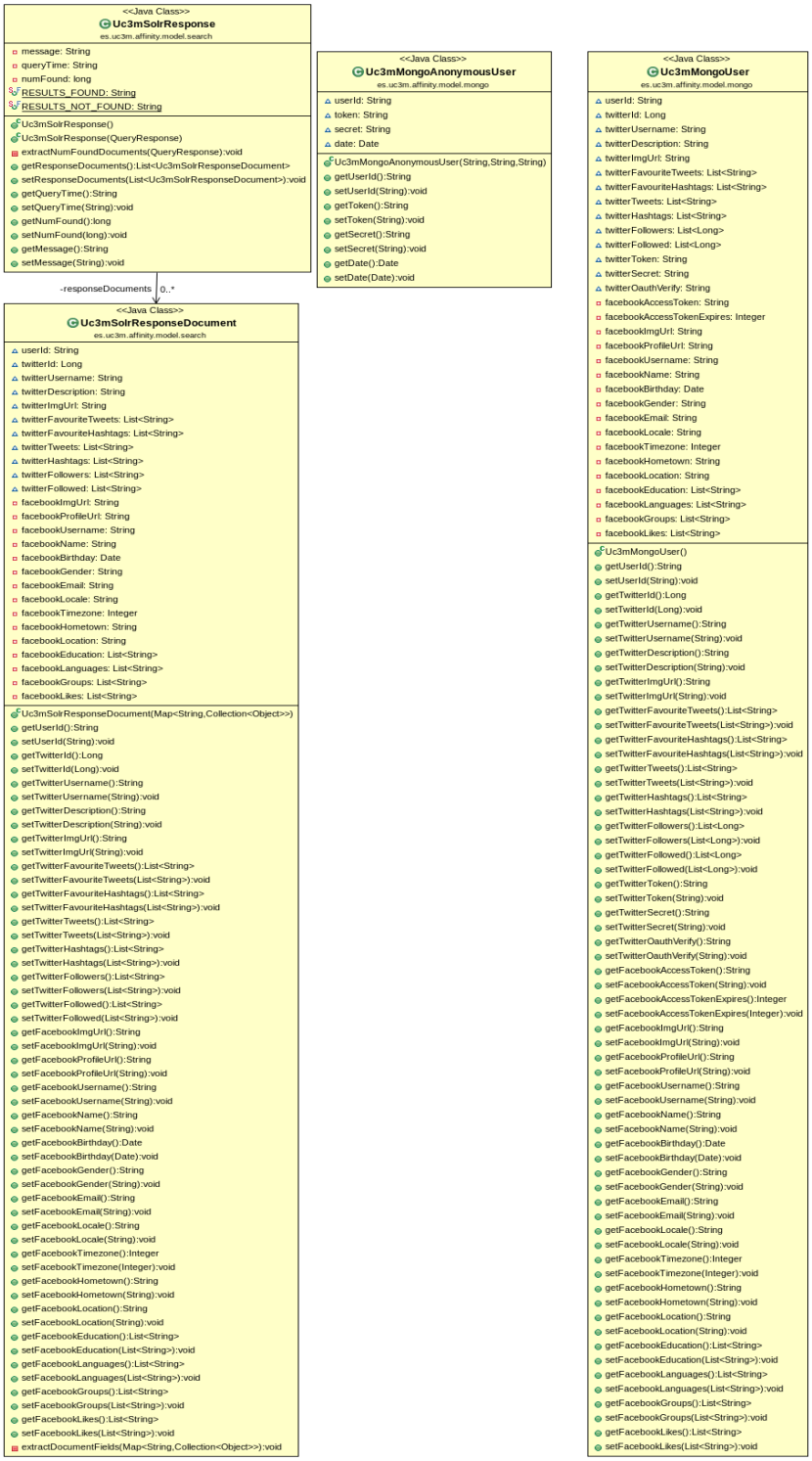

Figura 21: diagrama UML del paquete Models (segunda parte)

Las **clases Java** incluidas son las siguientes:

Clase	Descripción
Uc3mMongoUser	Clase Java que define el modelo de información asociado a los datos completos de un usuario, compuestos por la información extraída desde Twitter junto con la información extraída desde Facebook, incluyendo el identificador único de usuario. La información de este modelo será persistida por cada usuario en la colección de MongoDB `users.authenticated`.
Uc3mMongoAnonymousUser	Clase Java que define el modelo de información asociado a un usuario anónimo. La información de modelo será persistida en la colección de MongoDB `users.anonymous`.
Uc3mSolrResponse	Clase Java que define el modelo de información asociado a una respuesta completa de una operación implementada por *UniAffinity API*. Incluirá, entre otros datos de interés, un listado de `Uc3mSolrResponseDocument`.
Uc3mSolrResponseDocument	Clase Java que representa un documento Solr como unidad de información. Cada uno de estos documentos contendrá la información de un usuario que será devuelta como respuesta a una operación de *UniAffinity API*.

Tabla 9: descripción de clases contenidas en el paquete Models (segunda parte)

Models: servidor Solr

Por último, en la Figura 21 se incluyen las clases que modelan un servidor Solr a nivel lógico junto con las operaciones que puede llevar a cabo. En la Tabla 10 se describen las operaciones en el interfaz implicado y en la Tabla 11 las clases y su funcionalidad detallada.

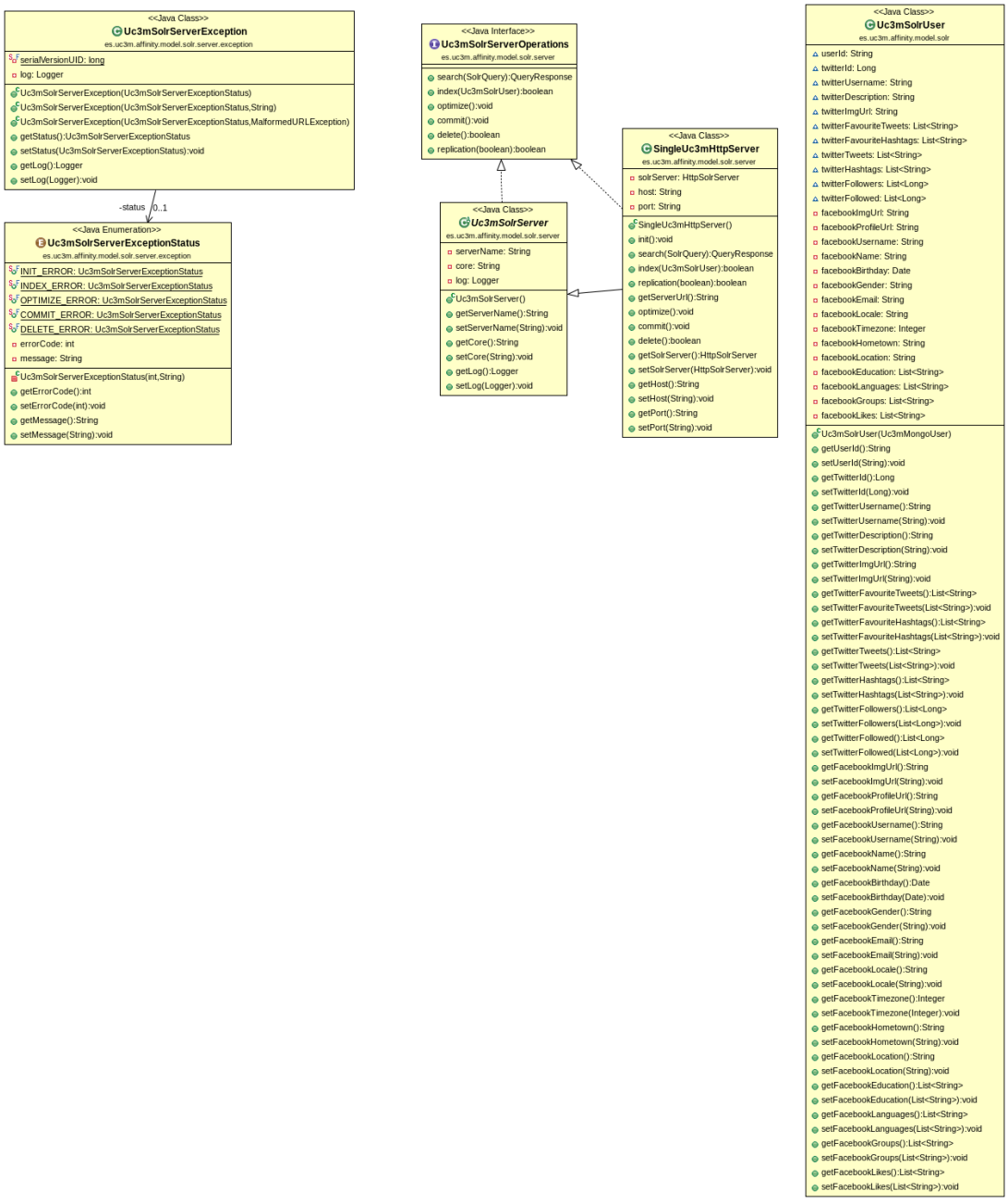

Figura 22: diagrama UML del paquete Models (tercera parte)

Los **interfaces** incluidos son las siguientes:

Interfaz	Descripción
Uc3mSolrServerOperations	Define todas las operaciones que un servidor Solr a nivel lógico puede llevar a cabo: búsquedas, indexaciones, etc. Este interfaz será implementado por la clase abstracta `Uc3mSolrServer`.

Tabla 10: descripción de interfaces contenidos en el paquete Models (tercera parte)

Las **clases Java** incluidas son las siguientes:

Clase	Descripción
Uc3mSolrServer	Clase Java `abstract` que define el modelo de información asociado a los datos necesarios para construir un servidor Solr a nivel lógico, incluyendo referencias a la máquina en la que está instalado, el puerto, el nombre del servidor Solr, etc. Implementa el interfaz Uc3mSolrServerOperations, pero no desarrolla la lógica de ninguno de los métodos que define `Uc3mSolrServerOperations`. La clase `Uc3mSolrServer` será extensible por otras clases para la definición de tipos particulares de servidores Solr siguiendo un patrón de diseño *Prototype*.
SingleUc3mSolrServer	Clase Java que hereda de `Uc3mSolrServer` define un modelo de servidor Solr *standalone* en el que se desarrolla toda la lógica de negocio en cada uno de los métodos que son definidos por `Uc3mSolrServerOperations`.

Uc3mSolrServerException	Clase Java que hereda de `Exception` para definir un tipo de excepción que el interfaz `Uc3mSolrOperations` define que será lanzada en caso de error de alguno de sus métodos definidos.
Uc3mSolrUser	Clase Java que define el modelo de datos asociado a la información de usuario que va a ser indexada en un servidor Solr. Cada una de las instancias de `Uc3mSolrUser` que sean indexadas representará un documento único en un servidor Solr.

Tabla 11: descripción de clases contenidas en el paquete Models (tercera parte)

4.3 Persistencia

4.3.1 Colecciones en MongoDB

Como sabemos, MongoDB almacenará persistentemente la información extraída del usuario. Además, el diálogo de autenticación de un usuario – nuevo o no – requiere un estado intermedio entre usuario no identificado y usuario registrado: usuario anónimo.

El estado de un usuario anónimo será compartido entre la *cookie* `uc3m-anonymous` y MongoDB. Dicho de otro modo, para que un usuario anónimo exista y pueda ser identificado como tal debe tener un registro en MongoDB con un identificador asociado y también una cookie `uc3m-anonymous` que incluirá dicho identificador. De lo contrario, la relación sería inexistente y no podría comprobarse su veracidad.

Dispondremos de una base de datos en MongoDB llamada `uc3m`. Recordemos que esta base de datos contendrá dos colecciones distintas: `users.anonymous`, donde almacenaremos los registros temporales de los usuarios anónimos, y `users.authenticated`, donde almacenaremos persistentemente los datos de los usuarios ya registrados.

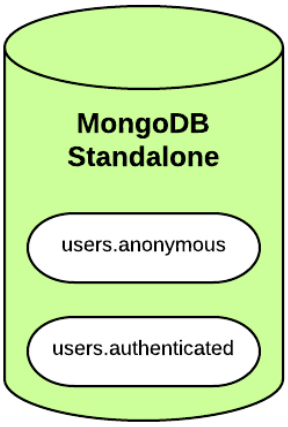

Figura 23: colecciones definidas para UniAffinity en MongoDB

En la colección `users.anonymous`, *UniAffinity Backend* almacenará la información relativa al inicio de un diálogo de autenticación por un usuario no reconocido por el sistema; es decir, sin *cookie* `uc3m`.

La información almacenada en la colección tendrá el siguiente formato:

```
{
        "_id" : "d9dde8f1-c2cb-4d32-9dce-36780455d785",
        "_class" : "es.uc3m.affinity.model.mongo.Uc3mMongoAnonymousUser",
        "token" : "MlUqlw4biKBAAcEbO6QeUajMaRWec4x6NkQQUpWbso",
        "secret" : "IXG0St54GaGRi704qogptNJihybhUTd8qVNpIIE0sw",
        "date" : ISODate("2013-09-15T09:14:33.850Z")
}
```

Código 2: ejemplo BSON en la colección users.anonymous

En el BSON anterior podemos encontrar los siguientes campos:

- **Id**: el identificador único creado por *UniAffinity*.

- **Class**: referencia a la clase Java de la que provienen los datos. Es necesario incluir esta referencia para que Java, a través de reflexión, pueda *serializar* y *deserializar* datos hacia y provenientes de MongoDB.

- **Secret**: el secreto asociado de la cuenta de la red social. Este campo se genera automáticamente desde la red social y tiene caducidad.

- **Token**: el *token* asociado a la cuenta de la red social del usuario. Este campo se genera automáticamente desde la red social y tiene caducidad.

- **Date**: fecha de inserción del registro en la colección `users.anonymous`.

En la colección `users.authenticated` almacenaremos la información al completo de un usuario registrado e identificado por el sistema. La información será tan extensa como lo sea la actividad del usuario en la red o redes sociales, por lo que el BSON resultante podría llegar a ser muy extenso.

A modo de ejemplo, analicemos un BSON reducido almacenado en MongoDB en la colección `users.authenticated` perteneciente a un usuario registrado a través de Twitter:


```
{
	"_id": "d9dde8f1-c2cb-4d32-9dce-36780455d785",
	"_class": "es.uc3m.affinity.model.mongo.Uc3mMongoUser",
	"twitterId" : NumberLong(280002418),
	"twitterUsername" : "lcappadev",
	"twitterDescription" : "Rover Software Engineer & Project
Manager. Stack: Solr, Lucene, Redis, MongoDB, Hadoop, Storm, Mahout,
Node.js, Spring, Bootstrap, REST, Cloud. ",
	"twitterImgUrl" :
"http://a0.twimg.com/profile_images/3585430331/2d1a8278462c603d8e81e03
bf7eecee8_normal.png",
	"twitterFavouriteTweets" : [
		"Aug 31: Node.js in Practice #manningbooks
http://t.co/TZNXmP9oNd"
	],
	"twitterFavouriteHashtags" : [
		"manningbooks"
	],
	"twitterTweets": [ "RT @TwitterEng: We just open sourced
@summingbird, streaming mapreduce with @scalding and @stormprocessor
https://t.co/iLgkUEiUA7"
	],
	"twitterHashtags" : [
		"NoETL",
		"NoSQL"
	],
	"twitterFollowers" : [
		NumberLong(813485449)

	],
	"twitterFollowed" : [
```

```
            NumberLong(287049865)
        ]
}
```

Código 3: ejemplo BSON en la colección users.authenticated

En el BSON anterior podemos encontrar los siguientes campos, que ya nos resultarán muy familiares:

- **Id**: el identificador único creado por *UniAffinity* que será asociado al usuario registra
- **Class**: referencia a la clase Java de la que provienen los datos. Es necesario incluir esta referencia para que Java, a través de reflexión, pueda *serializar* y *deserializar* datos hacia y provenientes de MongoDB.
- **TwitterId**: identificador numérico en Twitter asociado al usuario de Twitter.
- **TwitterUsername**: nombre de usuario en Twitter.
- **TwitterDescription**: descripción del usuario en su perfil de Twitter.
- **TwitterImgUrl**: URL que apunta a la imagen estática del usuario en su perfil de Twitter.
- **TwitterFavouriteTweets**: listado de los tweets marcados como favoritos del usuario en Twitter.
- **TwitterFavouriteHashtags**: etiquetas o *hashtags* contenidas en los tweets marcados como favoritos por el usuario en Twitter.
- **TwitterTweets**: texto íntegro de los últimos tweets del usuario.
- **TwitterHashtags**: etiquetas o *hashtags* contenidas en los últimos tweets del usuario.
- **TwitterFollowers**: identificadores numéricos de los usuarios que siguen al usuario en Twitter.
- **TwitterFollowed**: identificadores numéricos de los usuarios a los que el usuario sigue en Twitter.

4.3.2 Configuración en UniAffinity Backend

En *UniAffinity Backend* podremos configurar a cuál de datos MongoDB vamos a conectarnos. Es decir, tanto la máquina, puerto, nombre de la base de datos en MongoDB… son parametrizables en *UniAffinity Backend*.

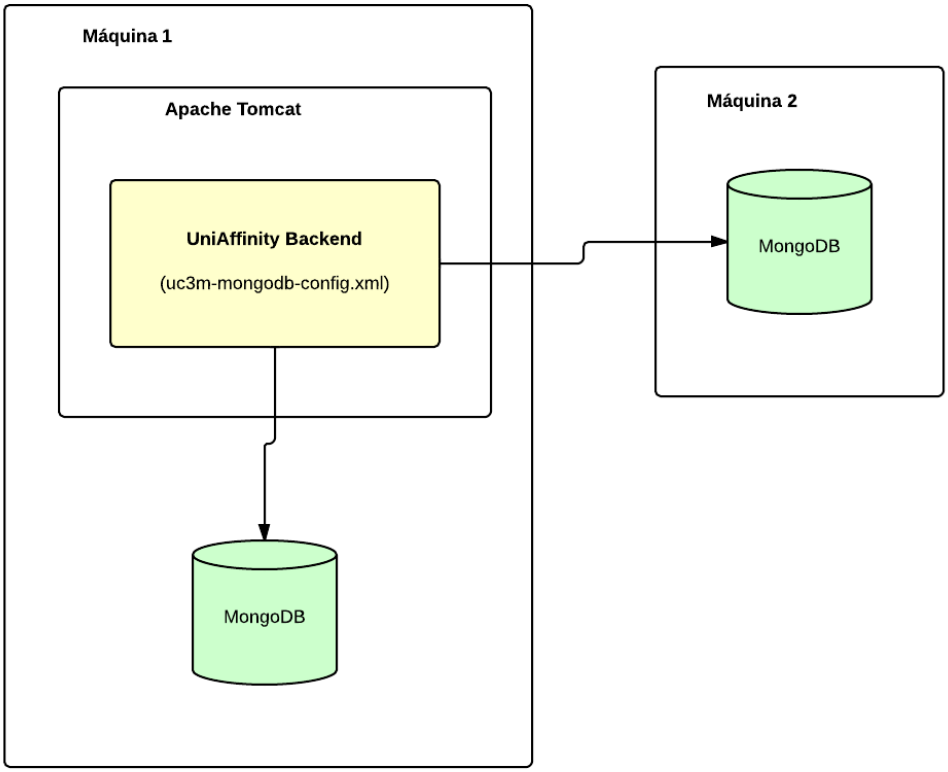

Figura 24: comunicación entre UniAffinityBackend y MongoDB

En la anterior imagen podemos observar dos modelos distintos de configuración para MongoDB en *UniAffinity Backend*. El primero de ellos, y más básico, consistiría en tener en una misma máquina servidora tanto *UniAffinity Backend* desplegado en un servidor Apache Tomcat, como la instancia de MongoDB para la persistencia de datos.

La segunda alternativa sitúa a MongoDB en una máquina servidora distinta a la de *UniAffinity Backend*. La comunicación entre módulos se hará a través del driver de comunicación Java para MongoDB, por lo que es posible adoptar esta estrategia más descentralizada.

El cliente de acceso a MongoDB a través del driver es configurado través de la gestión de dependencias de Spring. Podemos configurar nuestro conector de acceso a MongoDB en el

archivo de configuración `uc3m-mongodb-config.xml` para dar soporte a cualquiera de las dos alternativas de configuración de MongoDB propuestas.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:util="http://www.springframework.org/schema/util"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:mongo="http://www.springframework.org/schema/data/mongo"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
      http://www.springframework.org/schema/util
      http://www.springframework.org/schema/util/spring-util-3.1.xsd
      http://www.springframework.org/schema/data/mongo
      http://www.springframework.org/schema/data/mongo/spring-mongo-
1.1.xsd">

      <mongo:db-factory id="mongoDbFactory" host="localhost"
port="27017" dbname="uc3m" />

      <bean id="mongoTemplate"
class="org.springframework.data.mongodb.core.MongoTemplate">
            <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"
/>
      </bean>

</beans>
```

Código 4: contenido de uc3m-mongodb-config.xml

En el anterior archivo de configuración podemos comprobar cómo Spring define una conexión hacia MongoDB para un *host*, puerto y base de datos definida. Si quisiéramos modificar estos datos, o incluso incluir autenticación para acceder MongoDB, ése sería el lugar en el que configurarlo.

4.4 Motor de búsqueda

UniAffinity contará con su propio motor de Búsqueda basado en Apache Solr: *UniAffinity Solr Server*. *UniAffinity Solr Server* es un proyecto que incluye toda la configuración del Motor de

Búsqueda junto con otras utilidades que permiten una configuración y despliegue rápidos y eficaces.

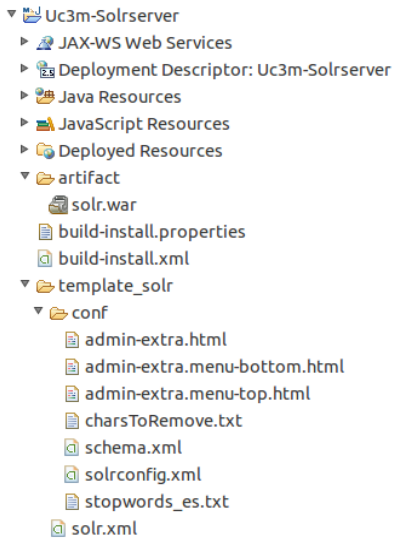

Figura 25: Organización de carpetas de UniAffinity Solr Server

Como podemos observar en la imagen, *UniAffinity SolrServer* incluye:

- Un servidor Apache Solr: `solr.war`. Su versión corresponde con la versión Apache Solr 4.3.0.

- Los diferentes archivos de configuración de un servidor Solr, a destacar:

    1. **Archivo solr.xml:** define cuántos *cores* tiene el servidor Solr configurado.

    2. **Archivo schema.xml:** define el esquema de datos del servidor Solr configurado.

    3. **Archivo solrconfig.xml:** define la configuración básica de un servidor Solr desde dos puntos de vista distintos: funcionalidades activadas o desactivadas y parámetros que pueden mejorar el rendimiento de Solr.

- Un instalador automático: script en Ant [47], *`build-install.xml`*, que permite el despliegue automático de un servidor Solr a partir de las propiedades que se definan en *`build-install.properties`*.

Una vez desplegado en un servidor Apache Tomcat determinado, la estructura de carpetas final en el directorio de instalación de Apache Tomcat quedará de un modo similar a éste:

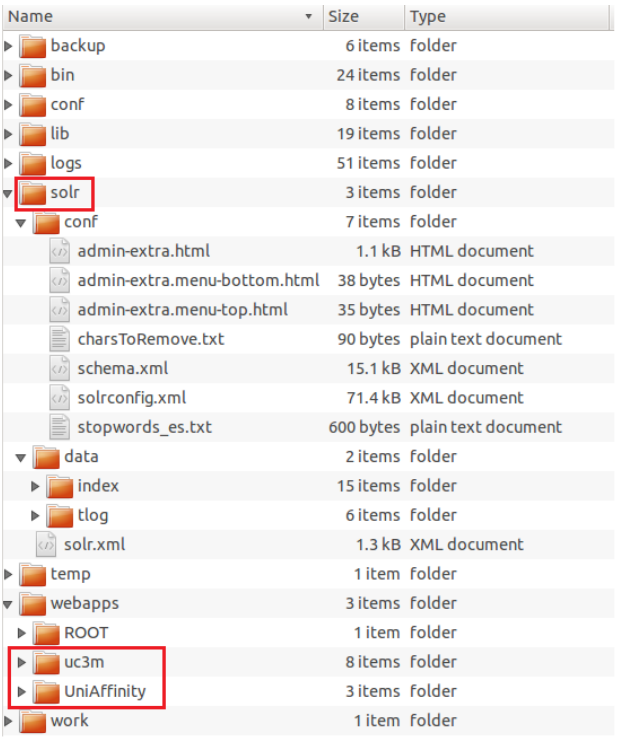

Figura 26: Instalación completa en Apache Tomcat

En la imagen anterior podemos encontrar los siguientes módulos o carpetas:

- **Solr:** en la carpeta "*solr*" se encontrarán todos y cada uno de los archivos de configuración necesarios para poder arrancar correctamente el servidor, así como los índices de datos que se vayan generando.

- **Uc3m:** será el nombre de nuestro *UniAffinity Solr Server* una vez haya sido desplegado en el servidor de aplicaciones.

- **UniAffinity Backend (Opcional)**: al igual que el servidor Solr, en la carpeta "*webapps*" podría desplegarse nuestra aplicación *UniAffinity Backend*. Recordemos que no es necesario que tanto el servidor *UniAffinity Solr Server* como *UniAffinity Backend* estén desplegados en el mismo Apache Tomcat, ni en la misma máquina servidora: podrían estar en diferentes Apache Tomcat y en diferentes servidores.

4.4.2 Configuración en UniAffinity Backend

Al igual que MongoDB, *UniAffinity Backend* permite indicar dónde encontrar el servidor Apache Solr al que realizar las consultas. Para ello *UniAffinity Backend* implementa una clase Java que

define qué parámetros debería incluir un servidor Solr, así como también una referencia a la clase Java (`HttpSolrServer`) del driver de conexión con Solr (`SolrJ`).

A la hora de configurar nuestro servidor Solr, y de un modo idéntico al que ocurriera con MongoDB, podríamos decidir tener el servidor en una máquina distinta a la que tiene desplegada *UniAffinity Backend*. Para ello, *UniAffinity Backend* permite definir los parámetros de configuración siguientes en el archivo de configuración `uc3m-meta-beans.xml`, que puede encontrarse en el interior del mismo proyecto:

```
      <bean id="singleUc3mHttpServer"
class="es.uc3m.affinity.model.solr.server.SingleUc3mHttpServer" init-
method="init">
           <property name="serverName" value="uc3m"/>
           <property name="core" value="users"/>
           <property name="host" value="localhost"/>
           <property name="port" value="8080"/>
    </bean>
```

Código 5: configuración de Solr en uc3m-meta-beans.xml

En el anterior apartado de configuración encontramos:

- **ServerName**: nombre del servidor Solr desplegado en Tomcat, que por defecto tomará el nombre de `uc3m`. El nombre será el mismo que adopte el servidor Solr `solr.war` al ser desplegado en el servidor Apache Tomcat.

- **Core**: parámetro de configuración de Solr. Un servidor Solr podría tener diferentes *cores*, que conceptualmente se asemejan a una colección de MongoDB. En nuestro caso sólo tendremos uno: `users`.

- **Host**: máquina donde estará alojado el servidor Solr. Recodemos que el servidor Solr podría estar en un servidor distinto al que tenga desplegado *UniAffinity Backend*, y aquí sería donde indicaríamos su procedencia.

- **Port**: puerto de acceso, si procede.

4.4.3 Configuración Solr

En la propia configuración del servidor Solr en sí mismo dispondremos de diferentes archivos de configuración que definan el comportamiento de nuestro servidor Solr, donde recordemos que los más relevantes son los archivos `solr.xml`, `schema.xml` y `solrconfig.xml`.

En el archivo `solr.xml` encontraremos la definición de cuántos *cores* tiene nuestro servidor Solr que, como sabemos, únicamente tendrá información relativa a los usuarios. Dado que no necesitamos más agrupaciones o cores de datos, declaramos como único core: `users`.

```
<solr persistent="false">
  <cores adminPath="/admin/cores" defaultCoreName="users">
    <core name="users" instanceDir="." />
  </cores>
</solr>
```

Código 6: archivo de configuración solr.xml

En el `schema.xml` definiremos los campos que contendrá nuestro índice de Solr. En estos campos almacenaremos la información de un usuario que pueda ser *buscable*, retornable o ambas, como ya vimos en el capítulo *2. Estado del arte* en el apartado de Apache Solr y Apache Lucene.

Se ha definido un tipo de campo que será empleado para todos aquellos campos que puedan ser empleados en operaciones de búsqueda por *UniAffinity API*, o que podrían serlo en un futuro. La definición del campo dentro del schema.xml es la siguiente:

```
<fieldType name="searchableTextTokenized" class="solr.TextField"
positionIncrementGap="100">
      <analyzer>
      <charFilter class="solr.MappingCharFilterFactory"
      mapping="charsToRemove.txt"/>
      <tokenizer class="solr.PatternTokenizerFactory"
      pattern="[\s:#@;\\.,\\(\\)\\{\\}\\/\\-]+" />
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.ASCIIFoldingFilterFactory"/>
      <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
      </analyzer>
</fieldType>
```

Código 7: definición del tipo de campo SearchableTextTokenized en schema.xml

Como recordaremos de apartados anteriores, Apache Solr permite definir una serie de operaciones a aplicar para extraer los términos finales de un texto de entrada, tanto en tiempo de búsqueda como en tiempo de indexación. Estas operaciones son:

1. Eliminación de caracteres especiales.

2. *Tokenización* a través de una expresión regular.

3. Transformación a minúsculas.

4. Transformación de términos con tildes, apóstrofes, etc. y normalización de los mismos a sus equivalentes sin esos caracteres.

5. Eliminación de términos duplicados.

Por último, dentro del archivo de configuración `solrconfig.xml` podemos definir la configuración de una de nuestras operaciones de búsqueda, `fulltext`, que a continuación veremos:

```
<requestHandler name="/fulltext" class="solr.SearchHandler">
 <lst name="defaults">
      <str name="hl">on</str> -->
      <str name="hl.fl">username description tweets</str>
      <str name="hl.requireFieldMatch">true</str>
   <str name="echoParams">explicit</str>
   <str name="defType">edismax</str>
   <str name="qf">twitterUsername^6.0 facebookUsername^6.0
twiterDescription^4.0 facebookName^4.0 twitterTweets^2.0
facebookGroups^2.0 facebookLikes^2.0</str>
   <str name="fl">*,score</str>
 </lst>
</requestHandler>
```

Código 8: configuración de operación fulltext en solrconfig.xml

La anterior configuración permite definir en el parámetro `qf` un listado de campos sobre los que buscar, junto con una serie de pesos asociados que son opcionales. Su función será la de buscar en todos estos campos la expresión de búsqueda introducida, poniendo especial interés en aquellos campos en los que el peso sea mayor a la hora de ordenar los resultados en la respuesta.

4.5 API

Las operaciones que *UniAffinity API* implementa son las siguientes:

1. **Autenticación con Twitter**

    a. **Path**: `/login/twitter`

    b. **Descripción**: petición que inicia el diálogo de autenticación con Twitter a través de *UniAffinity*. Implicará la aceptación de las condiciones de Twitter otorgando acceso a *UniAffinity App* a sus datos personales. Una respuesta correcta incluirá la *cookie* `uc3m-` que será necesaria incluir en posteriores peticiones al API - que acredita al usuario como usuario del sistema *UniAffinity*, dándole permiso a hacer uso del resto de operaciones de afinidad y búsqueda.

    c. **Parámetros**: ninguno.

2. **Autenticación con Facebook**

    a. **Path**: `/login/facebook`

    b. **Descripción**: petición que inicia el diálogo de autenticación con Facebook a través de *UniAffinity*. Implicará la aceptación de las condiciones de Twitter otorgando acceso a *UniAffinity App* a sus datos personales. Una respuesta correcta incluirá la *cookie* `uc3m-` que será necesaria incluir en posteriores peticiones al API - que acredita al usuario como usuario del sistema *UniAffinity*, dándole permiso a hacer uso del resto de operaciones de afinidad y búsqueda.

    c. **Parámetros**: ninguno.

3. **Affinity**

    a. **Path**: `/affinity/{start}/{rows}`

    b. **Descripción**: petición que, a partir del identificador de usuario que va implícito en la cookie uc3m, devuelve un listado de usuarios similares a uno mismo basándonos en el algoritmo de similitud de *UniAffinity*.

    c. **Parámetros**:
        i. *Start*: parámetro de paginación de resultados. Indica el punto de comienzo de la ventana de resultados.
        ii. *Rows*: parámetro de paginación de resultados. Indica el tamaño de la ventana de resultados, también llamado *offset*.

4. **Fulltext**

   a. **Path**: `/fulltext/{start}/{rows}/{text}`

   b. **Descripción**: petición que permite realizar una búsqueda de texto libre en el índice de los usuarios indexados. El texto libre indicado será buscado en los siguientes campos:

      - `TwitterUsername.`
      - `FacebookUsername.`
      - `TwiterDescription.`
      - `FacebookName.`
      - `TwitterTweets.`
      - `FacebookGroups.`
      - `FacebookLikes.`

      Pese a que se trata de una búsqueda de texto libre, *UniAffiniy* establece pesos de medición de relevancia a cada uno de estos campos, dándole más o menos importancia a los términos del texto libre indicado que en ellos aparezca para que los resultados obtenidos, al ser ordenados por relevancia basada en el algoritmo *tf-idf*, se vean afectados por el propio criterio de *UniAffinity*.

      Los pesos establecidos son los siguientes:

      - `TwitterUsername: 6.0.`
      - `FacebookUsername: 6.0.`
      - `TwiterDescription: 4.0.`
      - `FacebookName: 4.0.`
      - `TwitterTweets: 2.0.`
      - `FacebookGroups: 2.0.`

- `FacebookLikeS: 2.0.`

c. **Parámetros**:

i. *Start*: parámetro de paginación de resultados. Indica el punto de comienzo de la ventana de resultados.

ii. *Rows*: parámetro de paginación de resultados. Indica el tamaño de la ventana de resultados, también llamado *offset*.

iii. *Text*: texto libre de búsqueda.

5. Username

a. **Path**: `/username/{start}/{rows}/{text}`

b. **Descripción**: petición que realiza una búsqueda única de texto libre y exclusivamente en los nombres de usuario de Twitter y Facebook de los usuarios de *UniAffinity*.

c. **Parámetros**:

i. *Start*: parámetro de paginación de resultados. Indica el punto de comienzo de la ventana de resultados.

ii. *Rows*: parámetro de paginación de resultados. Indica el tamaño de la ventana de resultados, también llamado *offset*.

iii. *Text*: texto libre de búsqueda.

6. Tweets

a. **Path**: `/tweets/{start}/{rows}/{text}`

b. **Descripción**: petición que realiza una búsqueda de texto libre única y exclusivamente en los tweets de los usuarios de *UniAffinity* que hayan asociado una cuenta de Twitter.

c. **Parámetros**:

i. *Start*: parámetro de paginación de resultados. Indica el punto de comienzo de la ventana de resultados.

ii. *Rows*: parámetro de paginación de resultados. Indica el tamaño de la ventana de resultados, también llamado *offset*.

iii. *Text*: texto libre de búsqueda.

7. Description

a. **Path**: `/description/{start}/{rows}/{text}`

b. **Descripción**: petición que realiza una búsqueda de texto libre sobre aquellas descripciones de perfil de Twitter de todos los usuarios de *UniAffinity* que hayan vinculado una cuenta de Twitter.

c. **Parámetros**:

i. *Start*: parámetro de paginación de resultados. Indica el punto de comienzo de la ventana de resultados.

ii. *Rows*: parámetro de paginación de resultados. Indica el tamaño de la ventana de resultados, también llamado *offset*.

iii. *Text*: texto libre de búsqueda.

8. Likes

a. **Path**: `/likes/{start}/{rows}/{text}`

b. **Descripción**: petición que realiza una búsqueda de texto libre sobre todos aquellos *likes* de Facebook de todos los usuarios de *UniAffinity* que hayan vinculado una cuenta de Facebook.

c. **Parámetros**:

i. *Start*: parámetro de paginación de resultados. Indica el punto de comienzo de la ventana de resultados.

ii. *Rows*: parámetro de paginación de resultados. Indica el tamaño de la ventana de resultados, también llamado *offset*.

iii. *Text*: texto libre de búsqueda.

9. Groups

a. **Path**: `/groups/{start}/{rows}/{text}`

b. **Descripción**: petición que realiza una búsqueda de texto libre sobre todos aquellos grupos de Facebook de todos los usuarios de *UniAffinity* que hayan vinculado una cuenta de Facebook.

c. **Parámetros**:

i. *Start*: parámetro de paginación de resultados. Indica el punto de comienzo de la ventana de resultados.

ii. *Rows*: parámetro de paginación de resultados. Indica el tamaño de la ventana de resultados, también llamado *offset*.

iii. *Text*: texto libre de búsqueda.

Además, uno de los requisitos funcionales más importantes debe ser que todas estas operaciones deben ser accesibles únicamente para usuarios que se hayan registrado y autenticado en el sistema. Los datos almacenados son de especial importancia, y la privacidad de los mismos requiere respeto y prestar especial foco en materia de seguridad. Por esta razón *UniAffinity API* mantendrá como requisito y restricción funcional registro y autenticación previas para la ejecución de las operaciones del API.

Todas las respuestas frente a una operación de búsqueda en *UniAffinity API* (es decir, todas salvo las de autenticación con redes Sociales) vendrán en formato JSON y su formato será idéntico para todas ellas. En ella podremos encontrar:

- **Message**: indicará si se ha encontrado o no algún resultado. Podrá tomar los valores:

  1. `RESULTS FOUND`

  2. `RESULTS NOT FOUND`

- **NumFound**: número de usuarios encontrados en base a la petición lanzada contra *UniAffinity*.

- **QueryTime**: tiempo de ejecución de la respuesta en Solr. Útil para medición de rendimiento.

- **ResponseDocumentos**: listado de documentos (usuarios) resultantes. Todos y cada uno de estos documentos representarán a un usuario en *UniAffinity* e incluirán, a su vez,

todos sus parámetros de Facebook y Twitter que pertenezcan al perfil *buscable* de *UniAffiity*.

A continuación se adjunta un ejemplo completo de formato de JSON de respuesta de *UniAffinity* API. Se ha acortado el contenido de determinados campos de la respuesta por comodidad visual. Pese a todo, puede observarse cómo la información obtenida de los usuarios puede llegar a ser realmente completa e interesante de cara a posteriores análisis:


```
{
  • message: "RESULTS FOUND!",
  • numFound: 1,
  • responseDocuments:
  [
   {
    ▪ userId: "ee5cd521-c2ea-4821-be38-c989aa16ae58",
    ▪ twitterUsername: "lcappadev",
    ▪ twitterDescription: "Rover Software Engineer & Project
      Manager. Stack: Solr, Lucene, Redis, MongoDB, Hadoop,
      Storm, Mahout, Node.js, Spring, Bootstrap, REST, Cloud.",
    ▪ twitterImgUrl: "http://a0.twimg.com/profile_images/35854303
      31/2d1a8278462c603d8e81e03bf7eecee8_normal.png",
    ▪ twitterFavouriteTweets: [ "Scaling Pinterest: from 0 to 10s
      of billions of page views a month in
      two years http://t.co/4p2JHiFpgX" ],
    ▪ twitterFavouriteHashtags:[
      ▪ "Startups",
      ▪ "Entrepreneurs ],
    ▪ twitterTweets: [
      ▪ "@nathanmarz I agree 100%.",
      ▪ "Build a search engine using @neo4j Quite
        interesting! http://t.co/a4AfEdxAYr" ],
      ▪ twitterHashtags:[ "Postureo" ],
      ▪ twitterFollowers: [ "216040595", "12904842"],
      ▪ twitterFollowed: ["12904842", "23019136" ],
    ▪ facebookImgUrl: "https://graph.facebook.com/luiscappabanda/
      picture?type=large",
    ▪ facebookProfileUrl: "https://www.facebook.com/luiscappaband
      a",
    ▪ facebookUsername: "luiscappabanda",
    ▪ facebookName: "Luis Cappa Banda",
    ▪ facebookBirthday: 477097200000,
    ▪ facebookGender: "male",
    ▪ facebookEmail: "luiscappa@gmail.com",
    ▪ facebookLocale: "es_ES",
    ▪ facebookTimezone: 2,
```

```
        ▪ facebookHometown: "Madrid, Spain",
        ▪ facebookLocation: "Madrid, Spain",
        ▪ facebookEducation: ["Universidad Carlos III de Madrid" ],
        ▪ facebookLanguages: ["English","France" ],
        ▪ facebookGroups:  [ "Luis' Cocktails" ],
        ▪ facebookLikes: [ "Dragon Age", "Jeff Buckley", "Halo", ]

      }

    ],

    queryTime: "94ms"

}
```


Código 9: ejemplo de respuesta UniAffinity API

5. Historia del proyecto

5.1 Hitos

La historia del proyecto comprende un período de investigación, desarrollo, pruebas y documentación de **aproximadamente dos años**. Este período de tiempo es elevado, pero su dilatación en el tiempo se debe a motivos profesionales, y muy especialmente personales, que impidieron una dedicación a tiempo completo al proyecto. En un contexto mucho más favorable probablemente habría sido finalizado en un tercio de ese tiempo.

El listado o *roadmap* de grandes objetivos del proyecto, ordenados cronológicamente junto con su coste en horas y algunas observaciones interesantes, es el siguiente:

Año	Mes	Hito	Duración (horas)	Observaciones
2011	Septiembre	Investigación Apache Solr	50	Debido a las constantes actualizaciones de versión de Apache Solr en los últimos meses, muchos cambios que se produjeron entre versión y versión afectaban al modo de configurar un servidor Solr, por lo que la documentación existente para versiones recientes era bastante pobre.
2011	Octubre	Investigación MongoDB	40	Excelente documentación oficial para entender el funcionamiento de MongoDB. Realicé un curso de formación online de 10Gen, la empresa fundadora.

2011	Noviembre	Investigación Spring Framework	50	El exceso de documentación, muy distinta entre versiones de Spring, junto con la amplia variedad de maneras de configurar Spring para un mismo propósito, hizo que este hito implicara muchas horas de lectura y asentamiento de conocimientos.
2011	Diciembre	Investigación Twitter API	30	En este hito no sólo investigué el modo de interactuar con Twitter API, sino también la operativa a seguir en un proceso de autenticación Oauth. Además, traté de investigué qué librerías Java *opensource* existían que sirvieran de cliente o *connector* con Twitter API para poder integrarla en *UniAffinity*.
2012	Enero	investigación Facebook API	60	Muchas dificultades para configurar una aplicación en Facebook. Para poder acceder a la sección de Desarrolladores debía confirmar mi cuenta de Facebook con un código que decían enviar a mi teléfono móvil, pero jamás llegó. Finalmente, y tras mucho insistir por correo, lograron solucionar mi problema y me dieron acceso como Desarrollador.
2012	Febrero	Configuración e Instalación Apache Solr server	30	Configuración y despliegue iniciales sencillos. Sin embargo, cada cambio en el esquema de datos de Solr para hacer pruebas obligaba a reinstalar,

				hecho que ralentizó mi avance.
2012	Marzo	Instalación y configuración MongoDB	5	Instalación de paquete en sistema operativo Ubuntu. Muy sencillo.
2012	Abril	Instalación Apache Tomcat	1	Instalación de paquete en sistema operativo Ubuntu. Muy sencillo.
2012	Mayo	Esqueleto básico UniAffinity	80	Definir una estructura de capas sólida y coherente con una arquitectura Spring fue costoso, pero el resultado final fue bueno.
2012	Junio	Integración UniAffinity con Twitter API	45	La integración con Twitter fue sencilla gracias a la librería Twitter4J.
2012	Julio	Integración UniAffinity con Facebook API	80	No existe ninguna librería estable que sirva de cliente de Facebook, de modo que tuve que desarrollar yo mi propia aplicación cliente que enviara peticiones a Facebook API y que recogiera y formateara las respuestas JSON de su sistema.
2012	Agosto	Definición de algoritmo de similaridad UniAffinity	20	Este hito lo resolví conceptualmente de manera rápida, pues tras entender el funcionamiento del algoritmo *tf-idf* y asimilar las posibilidades que daban Twitter y Facebook en la recuperación de datos de usuarios, el modo en que relacionar usuarios con usuarios fue resuelto sin complicaciones. Sí que dediqué más tiempo a

				tareas de implementación.
2012	Septiembre	Definición de operaciones UniAffinity API	5	Una vez obtenidos y estructurados los datos de usuarios, decidir qué operaciones ofrecer desde *UniAffinity API* fue sencillo.
2012	Octubre	Implementación operaciones UniAffinity API	30	La implementación de las operaciones llevó algo más de tiempo, pues había que desarrollar SolrManager.java, un manager de operaciones de Solr que permita ejecutar las operaciones de *UniAffinity API*. La sintaxis de Solr para ejecutar consultas es compleja y poco intuitiva; sumado a esto, SolrJ, su driver de conexión, no es demasiado legible tampoco, algo que penalizó en tiempo a la hora de desarrollar este hito. Como punto positivo a destacar la gran comunidad de desarrolladores del proyecto Apache Solr, que siempre que tuve una duda me contestaron por correo en el mismo día con claridad.

2012	Noviembre	Tests y pruebas	60	Pruebas funcionales y corrección de errores.
2012	Diciembre	Instalador automático Apache Solr server	10	Debida a la compleja instalación de un servidor Solr, terminé por automatizar este proceso con un instalador construido con un *script* Ant sencillo.
2013	Enero	Documentación	50	Elaboración del índice de contenido junto con mi tutora.
2013	Febrero	Documentación	50	Estado del arte.
2013	Marzo	Documentación	50	Estado del arte.
2013	Abril	Documentación	50	Estado del arte.
2013	Mayo	Documentación	50	Descripción del sistema.
2013	Junio	Documentación	50	Descripción del sistema
2013	Julio	Documentación	50	Desarrollo del proyecto.
2013	Agosto	Documentación	50	Hitos, presupuesto y anexos.
2013	Septiembre	Documentación	50	Conclusiones, resúmenes, desarrollos futuros.

Tabla 12: planificación por hitos de UniAffinity

5.2 Análisis económico

Tras haber analizado el reto técnico y el esfuerzo necesario para la realización del proyecto, pasaremos a analizar la cuantía económica del mismo. Para ello se incluye un análisis detallado de, en primer lugar, el personal necesario para la consecución del proyecto; en segundo lugar, la inversión necesaria en equipos electrónicos, software, etc. Por último, se incluye también un porcentaje asociado a costes indirectos relativos a gastos poco cuantificables como la luz eléctrica, agua o la conexión a Internet.

5.2.1 Personal

El personal dedicado al desarrollo de *UniAffinity* se compone de un Desarrollador *Junior* y un Desarrollador *Senior*. Los recursos dedicados a cada una de esas vacantes serán, respectivamente, Luis Cappa banda y Florina Almenares Mendoza. Los costes incluyen los valores de IRPF y Seguridad Social.

Recurso	Categoría	Coste hombre/hora	Dedicación (horas)	Coste
Cappa Banda, Luis	Desarrollador Junior	26 €/h	1046	27196 €
Almenares Mendoza, Florina	Desarrolladora Senior	38 €/h	53	2014 €

TOTAL	29210 €

Tabla 13: coste de personal para el proyecto UniAffinity

5.2.2 Hardware

El coste de hardware va asociado a una amortización de cada uno de los dispositivos físicos empleados para la consecución de *UniAffinity*. La amortización a considerar será de 48 meses, y todos los costes incluyen IVA.

Concepto	Unidades	Precio (unidad)	Amortización (meses)	Uso (meses)	Coste para el proyecto
Laptop Toshiba core i7 8GB RAM	1	995 €	48	24	497,5 €

TOTAL	497,5 €

Tabla 14: coste de hardware para el proyecto UniAffinity

5.2.3 Software

A continuación se listan todo el software necesario para el desarrollo del proyecto *UniAffinity*. Muchos de ellos son gratuitos por tratarse de software libre (opensource), pero igualmente se

han incluido en el listado para tener un conocimiento completo de las herramientas y recursos a emplear en el desarrollo del proyecto.

Concepto	Unidades	Precio (unidad)	Coste para el proyecto
Microsoft Windows 7	1	129 €	129 €
Ubuntu 12.04	1	0	0
Eclipse IDE	1	0	0
Java EE 6	1	0	0
Microsoft Office	1	70 €	70 €

TOTAL	199 €

Tabla 15: coste de sofware para el proyecto UniAffinity

5.2.4 Presupuesto final

A continuación se detalla el presupuesto final del proyecto. Se ha considerado un 20% sobre el total derivado de los costes indirectos al proyecto asociados a gastos de luz, agua, Internet, etc.

- Coste del recurso designado al desarrollo del proyecto *UniAffinity* teniendo en cuenta el total de horas dedicadas y un coste estimado de horas/hombre.

Coste total del proyecto	29906,5 €
Costes indirectos (20%)	5981,3 €

COSTE TOTAL PROYECTO	35887,8 €

Tabla 16: coste total del proyecto UniAffinity

6. Conclusiones y trabajos futuros

6.1 Conclusiones

UniAffinity fue concebido como un paquete compuesto por el *Backend* y el *Frontend*, siendo el *Frontend* un Portal Web adaptado a móviles construido con Twitter Bootstrap que consumiera de *UniAffinity* API y permitiera:

5. Registrar usuarios a través de Twitter y Facebook con *UniAffinity* como pasarela.

6. Buscar usuarios similares a uno mismo.

7. Búsqueda de texto libre que aplicara a la información contenida en tweets, descripciones, grupos y *likes* de los usuarios.

8. Mostrar resultados de búsqueda con vistas previas de perfiles de usuarios incluyendo:

    a. Fotografía.
    b. Nombre.
    c. País.
    d. Localidad (opcional).
    e. Sexo (opcional).
    f. Edad (opcional).
    g. Descripción (opcional).

9. Chat en tiempo real entre usuarios en línea.

Sin embargo, la complejidad del desarrollo de la parte *Backend* hizo que el desarrollo del Portal Web tuviera que descartarse para el *Proyecto de Fin de Carrera*, quedando a la espera de una versión evolutiva de *UniAffinity* posterior.

Por este motivo, el desarrollo de un Portal Web se convierte en la línea de desarrollo futuro más aconsejable y con más sentido, y probablemente podría considerarse como tema a abordar en un *Proyecto de Fin de Carrera*.

Pese a todo, el desarrollo de todo el *Backend* de *UniAffinity* se considera exitoso, pues todos los objetivos fijados para el proyecto se han cumplido. A destacar:

- Estudio de las tecnologías *opensource* más avanzadas en temática de almacenamiento no relacional capaces de ofrecer alto rendimiento en operaciones de escritura y lectura distribuidas.

- Diseño de una arquitectura modular y escalable que permite a entornos de baja carga (pocos usuarios concurrentes) y ampliarse de una manera sencilla en entornos de alta carga (muchos usuarios concurrentes).

- Implementación de un módulo de autenticación con redes sociales adaptable a cualquier desarrollo Web que, además, permite ofrecer información de los usuarios.

- Sistema de recomendación basado en búsquedas que permite recomendar usuarios sin entrenamiento resolviendo consultas con gran rapidez.

- Desarrollo de un HTTP/S API abierto al uso de un gran abanico de clientes accesible a través de autenticación. La autenticación se lleva a cabo a través de Facebook y Twitter, adaptando *UniAffinity Backend* a sus implementaciones del protocolo OAuth 2.0 y OAuth 1.0, respectivamente.

6.2 Líneas de trabajo futuras

6.2.1 Clave de identificación por cliente

Sería realmente interesante incluir la figura del identificador del cliente, no sólo la del usuario, para que las peticiones a *UniAffinity API* no sólo tuvieran que estar autenticadas con el sistema, por lo que incluirían la cookie uc3m en él, sino que también tuvieran que incluir una clave de acceso a *UniAffinity API* que fuera única por cada aplicación cliente.

Este desarrollo permitiría la apertura e instalación de *UniAffinity* como servicio en la nube (SaaS) donde cada cliente tendría su clave de acceso y podría integrar *UniAffinity* en su sistema, ya sea una aplicación móvil, un portal Web o cualquier otro software que desee consumir de su API.

6.2.2 Aplicación móvil

La movilidad de los datos es un hecho. La expansión del uso de Internet en terminales y dispositivos móviles, ya sean teléfonos inteligentes o tabletas, ha convertido el desarrollo de aplicaciones móviles en uno de los negocios tecnológicos más rentables en la actualidad.

Poder contar con una aplicación móvil de *UniAffinity* situaría a la *Universidad Carlos III* en una posición diferenciadora con respecto al resto de universidades públicas de la Comunidad de Madrid. Incluyendo las mismas funcionalidades que el Portal Web descrito anteriormente, permitiría a los alumnos estar permanentemente conectados entre sí fomentando relaciones personales y ayudándoles a conocer gente común a ellos, algo que alumnos venidos de fuera de Madrid probablemente agradecerían.

Con todo el atractivo y las funcionalidades descritas para el Portal Web, si bien éste está adaptado a móviles, el desarrollo de una aplicación móvil podría incluir otras funcionalidades extra realmente potentes y diferenciadoras con respecto al Portal Web:

1. **Geolocalización**: utilizando el GPS (*Global Positioning System*) del móvil podríamos saber en qué latitud y longitud se encuentra el alumno en concreto, y uno de los criterios de filtrado de búsqueda de *UniAffinity API* podría ser ofrecer resultados de búsqueda que obedezcan a la lógica de la búsqueda, pero que además estén cerca del usuario.

2. **Notificaciones nativas**: podría ser interesante emplear notificaciones nativas de Android o iOS para enviar alertas al móvil siempre que un usuario esté cerca de algún otro, cuando alguien visite nuestro perfil, etc.

6.2.3 Análisis estadístico poblacional

Los datos recogidos son de gran valor para un estudio estadístico que, por ejemplo, permitan conocer con más detalle los gustos, aficiones, hábitos… de la comunidad de usuarios registrados. Uno de los fines más prometedores de este estudio podría ser la segmentación poblacional para campañas de mercadotecnia y para perfilado de usuarios dentro de un e-*commerce*.

Un segundo enfoque de perfil investigador podría partir de la propia *Universidad Carlos III*. Si bien la Universidad cuenta con una base de datos registrada de todos los alumnos que incluye sexo, edad, localidad de nacimiento, etc., estos datos personales básicos podrían verse cruzados y enriquecidos por aquellos que se extraigan de Twitter y Facebook para ampliar y mejorar la base de conocimiento total.

Este conocimiento permitiría, a futuro, llevar a cabo actividades y eventos universitarios que encajaran con los gustos generales de la población de alumnos, así como estudiar la progresión de las aficiones, actividad en redes sociales, etc. de todos ellos a lo largo del tiempo.

Referencias

[1] Twitter API. Disponible en: http://dev.twitter.com (Septiembre 2013)

[2] Twitter Storm. Disponible en: http://storm-project.net/ (Septiembre 2013)

[3] Twitter Bootstrap. Disponible en: http://getbootstrap.com/2.3.2/ (Septiembre 2013)

[4] Protocolo OAuth 1.0. Disponible en: http://tools.ietf.org/html/rfc5849 (Septiembre 2013)

[5] Google Play. Disponbile en: https://play.google.com/store (Septiembre 2013)

[6] Apple iTunes Store. Disponbile en: http://www.apple.com/itunes/ (Septiembre 2013)

[7] Protocolo OAuth 2.o. Disponible en: http://oauth.net/2/ (Septiembre 2013)

[8] Facebook Graph. Disponible en: https://developers.facebook.com/docs/reference/api/ (Septiembre 2013)

[9] HTTP. Disponible en: http://es.wikipedia.org/wiki/Hypertext_Transfer_Protocol (Septiembre 2013)

[10] SQL. Disponible en: http://www.w3schools.com/sql/ (Septiembre 2013)

[11] Facebook Graph Search. Disponible en: https://www.facebook.com/about/graphsearch (Septiembre 2013)

[12] RFC-5849. Disponible en: http://tools.ietf.org/html/rfc5849 (Septiembre 2013)

[13] Google AuthSub. Disponible en: https://developers.google.com/accounts/docs/AuthSub (Septiembre 2013)

[14] Yahoo BBAuth. Disponible en: http://developer.yahoo.com/auth/ (Septiembre 2013)

[15] Flickr API. Disponible en: http://www.flickr.com/services/api/ (Septiembre 2013)

[16] HMAC. Disponible en: http://www.ietf.org/rfc/rfc2104.txt (Septiembre 2013)

[17] SHA-256. Disponible en: http://tools.ietf.org/html/rfc5754 (Septiembre 2013)

[18] Base64. Disponible en: http://es.wikipedia.org/wiki/Base64 (Septiembre 2013)

[19] Apache Solr. Disponible en: http://lucene.apache.org/solr/ (Septiembre 2013)

[20] Apache Foundation. Disponible en: http://www.apache.org/ (Septiembre 2013)

[21] Apache Tomcat. Disponible en: http://tomcat.apache.org/ (Septiembre 2013)

[22] Jetty. Disponible en: http://www.eclipse.org/jetty/ (Septiembre 2013)

[23] SolrJ. Disponible en: http://wiki.apache.org/solr/Solrj (Septiembre 2013)

[24] Apache Lucene. Disponible en: http://lucene.apache.org/core/ (Septiembre 2013)

[25] Algoritmo Tf-Idf. Disponible en: http://es.wikipedia.org/wiki/Tf-idf (Septiembre 2013)

[26] PageRank. Disponible en: http://es.wikipedia.org/wiki/PageRank (Septiembre 2013)

[27] MySQL. Disponible en: http://www.mysql.com/ (Septiembre 2013)

[28] SQLite. Disponible en: http://www.sqlite.org/ (Septiembre 2013)

[29] PostgreSQL. Disponible en: http://www.postgresql.org (Septiembre 2013)

[30] Cassandra. Disponible en: http://cassandra.apache.org/ (Septiembre 2013)

[31] Redis. Disponible en: http://redis.io/ (Septiembre 2013)

[32] MongoDB. Disponible en: http://www.mongodb.org/ (Septiembre 2013)

[33] CouchDB. Disponible en: http://couchdb.apache.org/ (Septiembre 2013)

[34] Neo4J. Disponible en: http://www.neo4j.org/ (Septiembre 2013)

[35] Javascript. Disponible en: http://es.wikipedia.org/wiki/JavaScript (Septiembre 2013)

[36] BSON. Disponible en: http://bsonspec.org/ (Septiembre 2013)

[37] JSON. Disponible en: http://www.json.org/ (Septiembre 2013)

[38] Spring Framework. Disponible en: http://projects.spring.io/spring-framework/ (Septiembre 2013)

[39] JavaBeans. Disponible en: http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html (Septiembre 2013)

[40] J2EE. Disponible en: http://docs.oracle.com/javaee/ (Septiembre 2013)

[41] SourceForge. Disponible en: http://sourceforge.net/ (Septiembre 2013)

[42] RMI. Disponible en: http://projects.spring.io/spring-framework/ (Septiembre 2013)

[43] JMX. Disponible en: http://docs.oracle.com/javase/tutorial/jmx/overview/ (Septiembre 2013)

[44] TCP/IP. Disponible en: http://es.wikipedia.org/wiki/Modelo_TCP/IP (Septiembre 2013)

[45] Apache Maven. Disponible en: http://maven.apache.org/ (Septiembre 2013)

[46] Apache Mahout. Disponible en: http://mahout.apache.org/ (Septiembre 2013)

[47] Apache Ant. Disponible en: http://ant.apache.org/ (Septiembre 2013)

Bibliografía

"Lucene in Action, 2nd Edition", Erik Et Al Hatcher. Editorial Manning (28 de Julio del año 2010)

"Apache Solr 4 Cookbook", Rafal Kuc. Editorial Packt Publishing (25 de enero del año 2013)

"Spring in Action, 3rd Edition", Wall. Editorial Manning (1 de Julio del año 2011)

"Data Mining: practical Machine Learning tools and techniques, 3rd Edition", Ian H.Witten, Ejbe Frank, Mark A. Hall. Editorial Morgan Kaufmann (3 de Febrero del año 2011)

"MongoDB in Action", Kyle Banker. Editorial Manning (1 de Enero del año 2012)

Apéndices

A. Manual de Instalación

1. En un entorno Linux Ubuntu, instalar Apache Tomcat ejecutando en línea de comandos:

```
sudo apt-get install tomcat7
```

2. Instalar MongoDB ejecutando en línea de comandos:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
7F0CEB10

echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart
dist 10gen' | sudo tee /etc/apt/sources.list.d/mongodb.list

sudo apt-get update

sudo apt-get install mongodb-10gen
```

3. Instalamos Maven ejecutando en línea de comandos:

```
sudo apt-get install maven
```

4. Compilamos *UniAffinity*:

```
cd /home/username/uniaffinity/UniAffinity
mvn –e install
```

5. Desplegamos `uniaffinity.war`, resultante del proceso de compilación anterior, en Apache Tomcat:

```
sudo cp …/UniAffinity/target.uniaffinity.war /var/lib/tomcat7/webapps
```

6. Editamos el fichero de configuración de *UniAffinity Solr* Server llamado build-install.properties, que encontraremos en:

```
/home/username/uniaffinity/UniAffinitySolrServer/build-
install.properties
```

El fichero de configuración tendrá el siguiente aspecto:

```
####################################
###### DEVELOPMENT ENVIRONMENT ######
####################################

   # SOLR SERVERS NAME.
   SERVER_NAME=uc3m

   # TOMCAT WEBAPP DEPLOYMENT FOLDER PATH.
   TOMCAT_WEBAPPS=/var/lib/tomcat7/webapps

   # SOLR CONFIGURATION FOLDER PATH.
   SOLR_HOME=/var/lib/tomcat7/solr
```

En este fichero de configuración tendremos que indicar:

- El nombre de nuestro servidor Solr que, por defecto, será "uc3m".
- Dónde instalaremos el servidor Solr que, por defecto, será la carpeta "*webapps*" de Apache Tomcat.
- Dónde almacenaremos los archivos de configuración del Servidor Solr y su índice de datos. Por defecto, y también por comodidad, lo situaremos en el directorio de instalación de Tomcat.

7. Nos situamos en la raíz de *UniAffinity Solr Server* e instalamos el Servidor Solr ejecutando

```
sudo ant -f build-install.xml installSolr
```

8. Finalmente, arrancamos Apache Tomcat y observamos sus trazas de log para comprobar que el despliegue del servidor Solr y de *Uniaffinity* sean correctos. Ejecutamos el siguiente comando en línea de comandos:

```
sudo service tomcat7 start && sudo tail -f
      /var/log/tomcat7/catalina.out
```

B. Desarrollo de Apps

Twitter

Aunque de acceso libre y gratuito, algunos servicios que ofrece Twitter a la comunidad de desarrolladores están restringidos en uso para evitar su abuso y explotación indebida. En particular, y como se detallará más adelante, el uso del API de Twitter está restringido en base a unos credenciales de usuario. Basta con estar registrado en Twitter para poder crear una *App* y así obtener las claves necesarias para hacer uso de los servicios de Twitter.

Si deseamos crear una nueva Aplicación basta con dirigirnos al [Sitio de Desarrolladores de Twitter] y acceder al menú de creación de aplicaciones:

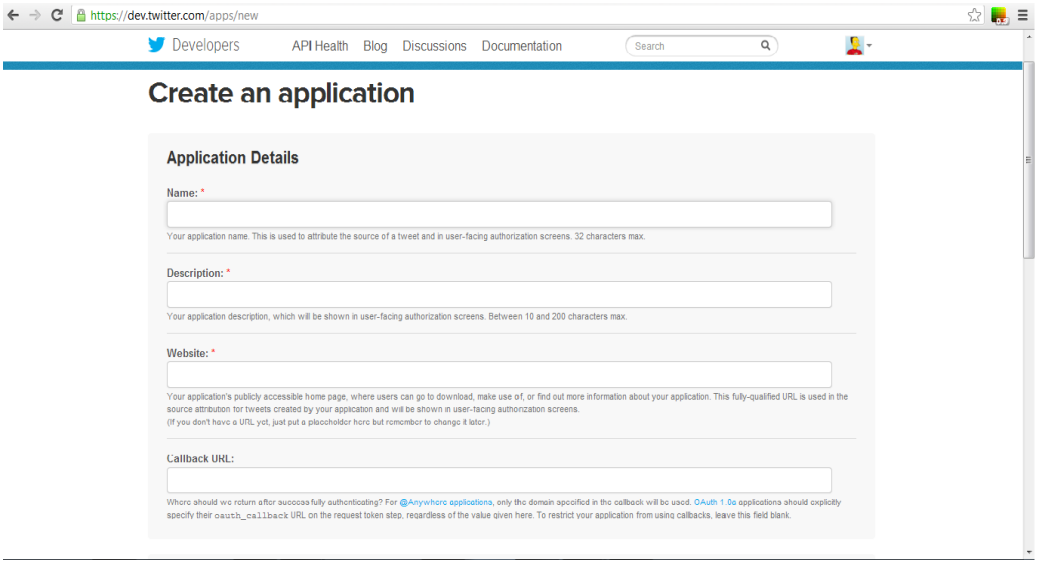

Figura 27: formulario de registro de Twitter Apps.

El formulario de registro es breve y sencillo, siendo de obligado cumplimiento indicar los siguientes campos:

- **Nombre**: el nombre de tu aplicación. En nuestro caso, el nombre elegido es "*UniAffinity*".
- **Descripción**: detalle de las características de funcionales y de uso de nuestra *App*.
- **Website**: URL de nuestro portal Web, si procede.

Fundamentalmente existe un solo tipo de *App*, pero en función del uso que se quiera dar de la información que podemos extraer de Twitter como fuente de contenidos podemos categorizarlas en tres categorías de uso:

- **Sólo lectura:** únicamente podremos acceder a la lectura del contenido proveído por Twitter (*tweets*).

- **Lectura y escritura:** podremos interactuar con el contenido, tanto leyendo como añadiendo nuevo contenido (*tweets*).

- **Lectura, escritura y acceso a mensajes privados:** la aplicación podrá interactuar con el contenido añadiendo y leyendo, pero también accediendo a los mensajes privados del usuario que haga uso de la *App*. El usuario estará dando consentimiento explícito y en todo momento sabrá que estas operaciones, delicadas en términos de privacidad a priori, son llevadas a cabo.

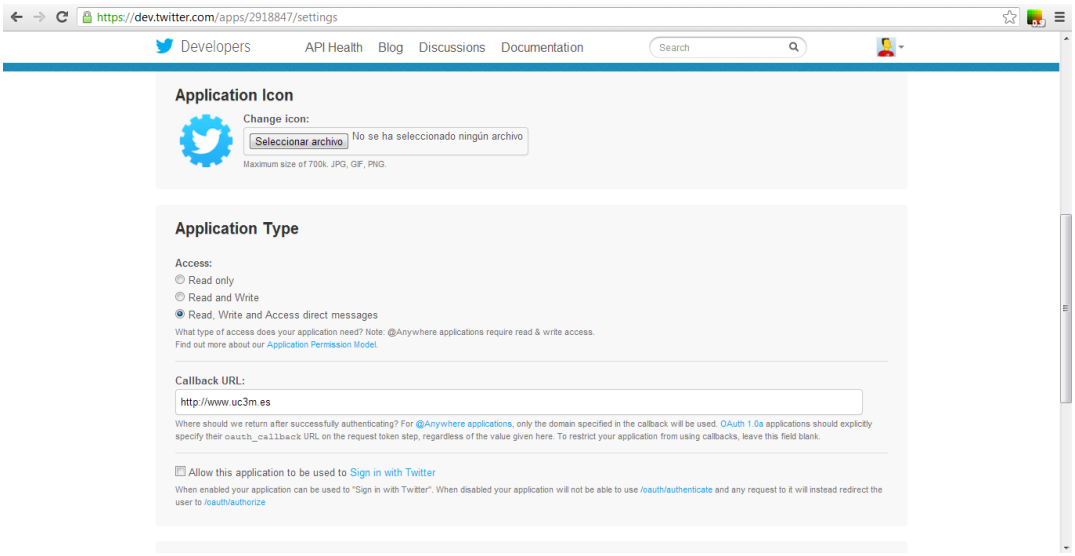

Figura 28: tipos de Twitter Apps

Una vez definidos los parámetros de configuración básicos para nuestra *App* ya podemos acceder a los cuatro credenciales de usuario que nos permiten identificarnos con los servicios de Twitter al hacer uso de ellos.

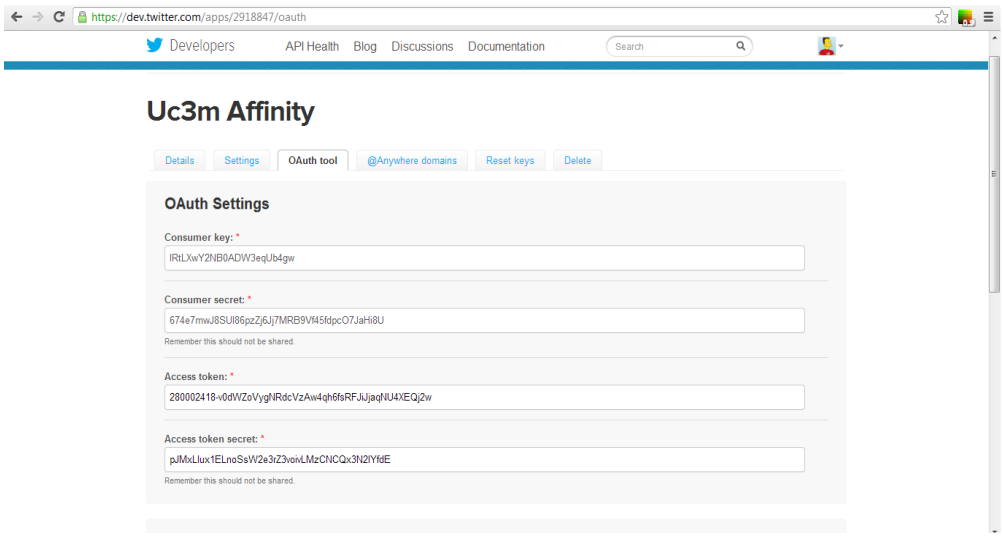

Figura 29: credenciales de acceso

Facebook

Las oportunidades para el desarrollo de *Apps* en Facebook son enormes, pero sólo podremos acceder a estos servicios y utilidades si nos registramos en el Portal de Desarrolladores. Para ello, debemos habernos registrado previamente en Facebook como usuarios. Adicionalmente a ambos registros deberemos:

1 Validar nuestra cuenta de desarrollador incluyendo nuestro número de teléfono móvil entre nuestros datos. Facebook nos enviará un código de confirmación y deberemos introducirlo para finalizar el registro completo.

2 Si no deseamos dar nuestro número de teléfono, Facebook ofrece la posibilidad de indicar nuestro número de tarjeta de crédito para finalizar el registro.

Una vez finalizado este proceso de registro, podremos acceder a crear una nueva Facebook App que nos permitirá tener acceso a las claves de identificación necesarias para poder interactuar con Facebook Graph y el resto de servicios disponibles.

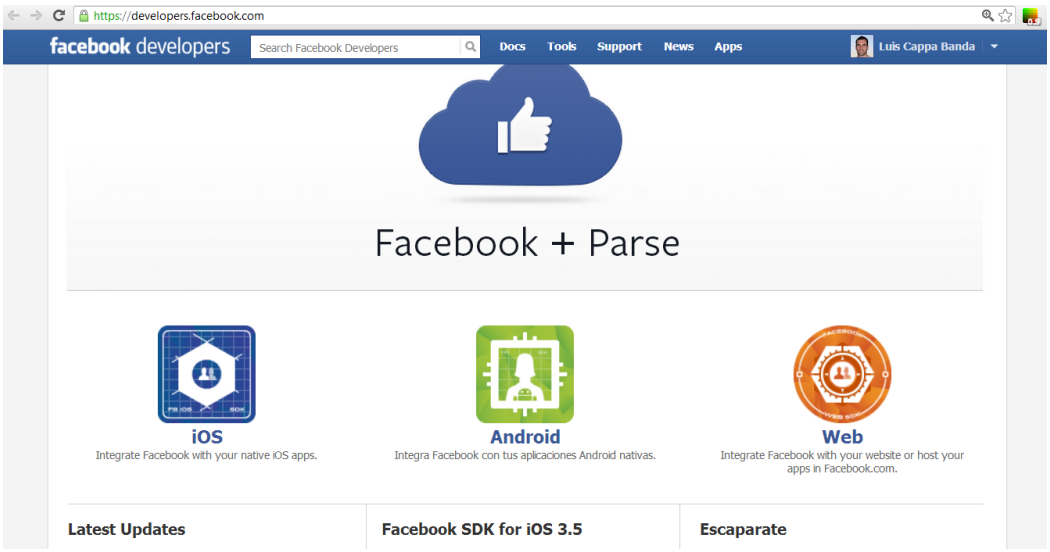

Figura 30: página principal del portal de desarrolladores de Facebook

Para crear una nueva Aplicación necesitamos introducir nuevos datos de configuración de la misma. Aunque las opciones son múltiples, y la mayoría no abarcar el cometido de este documento técnico, bastará con que indiquemos el nombre de la aplicación, si deseamos servicio de hospedaje y un *namespace* o espacio de nombres que definirán nuestras llamadas a Facebook Graph. Este espacio de nombres conviene ser igual o similar al nombre de nuestra aplicación, por comodidad.

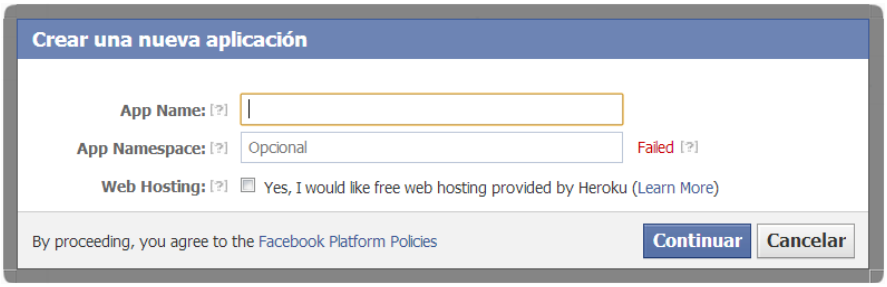

Figura 31: formulario de creación de nueva App

Una vez definidos estos parámetros básicos iniciales, podremos terminar de configurar nuestra App para tener acceso a las claves de servicio de Facebook Graph. Por defecto la nueva App aparece con el modo "*sandbox*" o modo de prueba. El ejemplo adjunto aparece con el modo *sandbox "Sitio web con inicio de sesión en Facebook"* desactivado y con la App configurada con la opción marcada. Para ello hemos facilitado un dominio ficticio, http://www.uniaffinity.com, requisito indispensable para finalizar la configuración de esta *App*.

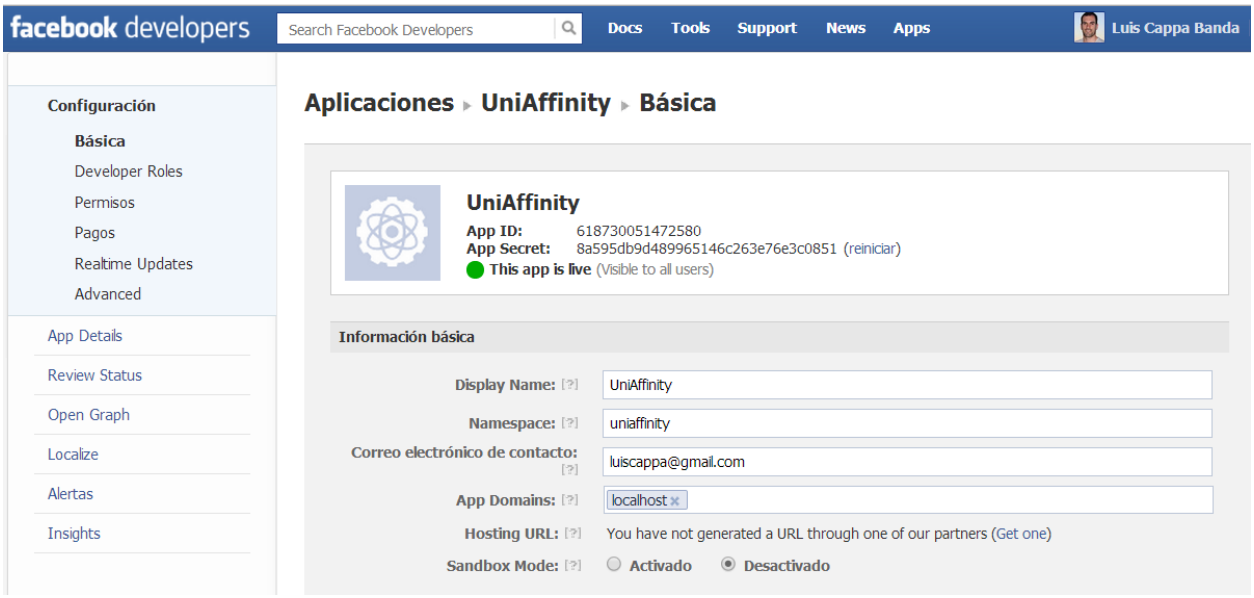

Figura 32: *formulario de configuración detallada de una Facebook App*

En la imagen anterior también aparecen las claves necesarias para establecer diálogos con Facebook Graph API: *App ID* y *App Secret*. Estos dos *tokens* van intrínsecamente ligados a nuestra *App*, si bien en cualquier instante podríamos decidir renovarlos, reiniciándolos.

C. Glosario de términos

App: abreviatura en inglés referida a aplicación comúnmente empleada en el contexto de las aplicaciones móviles.

Aprendizaje máquina: rama de la Inteligencia Artificial cuyo objetivo es desarrollar técnicas que permitan a las computadoras aprender

CTO: siglas de 'Chief Technical Officer', en inglés, referida al puesto de líder técnico de una empresa.

***Data Mining*:** terminología en inglés asociada a la extracción de datos en bruto de fuentes como la Web, grandes bases de datos no estructuradas, etc. con el fin de procesar la información, cribarla y extraer patrones que arrojen una información más concreta, refinada y exacta.

***Information Retrieval*:** (sinónimo de Data Mining). Terminología en inglés asociada a la práctica de recuperación de información, típicamente proveniente de documentos o textos electrónicos.

JAR: formato de archivo comprimido que incluye código Java compilado.

Mercadotecnia: ciencia relacionada con Marketing y con Publicidad que estudia el comportamiento de los usuarios para dirigir campañas orientadas a los segmentos poblacionales que estos ocupan.

***Mobile Revolution*:** revolución tecnológica asociada al emergente y masivo uso de teléfonos inteligentes a partir del año 2012.

NoSQL: 'Not Only SQL', en inglés. Término o abreviatura referida al compendio de bases de datos no relacionales tales como MongoDB, Cassandra, CouchDB, Redis, etc.

SaaS: siglas de 'Software as a Service', en inglés, referida al ofrecimiento de servicios de software en la nube.

SDK: kit de desarrollo de software.

***Stopwords*:** listado de palabras que son eliminadas de un contenido de texto determinado, típicamente aplicadas al comienzo de alguna cadena de procesado de texto.

TTL: siglas de 'Time To Live', en inglés, equivalentes a tiempo de vida de un determinado valor, registro, etc.

WAR: formato de archivo comprimido que incluye una aplicación Web Java desplegable en un servidor de aplicaciones.